



Red Hat Enterprise MRG 2 Realtime Referenzhandbuch

Referenzhandbuch für die Realtime-Komponente von Red Hat Enterprise
MRG
Ausgabe 3

Lana Brindley

Alison Young

Cheryn Tan

Referenzhandbuch für die Realtime-Komponente von Red Hat Enterprise MRG Ausgabe 3

Lana Brindley
Red Hat Engineering Content Services

Alison Young
Red Hat Engineering Content Services

Cheryn Tan
Red Hat Engineering Content Services
cheryntan@redhat.com

Rechtlicher Hinweis

Copyright © 2012 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Zusammenfassung

Dieses Buch enthält Referenzmaterial für die MRG Realtime Komponente von Red Hat Enterprise MRG, der Plattform für verteiltes Rechnen. Installationsanweisungen finden Sie im MRG Realtime Installationshandbuch. Informationen zur Optimierung finden Sie im MRG Realtime Optimierungshandbuch.

Inhaltsverzeichnis

Vorwort	4
1. Dokumentkonventionen	4
1.1. Typografische Konventionen	4
1.2. Konventionen für Seitenansprachen	6
1.3. Anmerkungen und Warnungen	7
2. Hilfe bekommen und Feedback geben	8
2.1. Brauchen Sie Hilfe?	8
2.2. Wir freuen uns auf Ihr Feedback!	8
Teil I. Hardware	9
Kapitel 1. Prozessorkerne	10
1.1. Caches	10
1.2. Interconnects	10
Kapitel 2. Speicherzuweisung	11
2.1. Demand Paging	11
2.2. Die Verwendung von mlock zur Vermeidung von Speicherfehlern	13
Kapitel 3. Hardware-Interrupts	16
3.1. Level-Signalled Interrupts	17
3.2. Message-Signalled Interrupts	17
3.3. Non-maskable Interrupts	18
3.4. System Management Interrupts	18
3.5. Advanced Programmable Interrupt Controller	19
Teil II. Applikationsarchitektur	20
Kapitel 4. Threads und Prozesse	21
Kapitel 5. Prioritäten und Richtlinien	23
Kapitel 6. Affinität	24
6.1. Verwendung des taskset-Befehls zur Einstellung der Prozessoraffinität	25
6.2. Verwendung des sched_getaffinity()-Systemaufrufs zur Einstellung der Prozessoraffinität	26
Kapitel 7. Thread-Synchronisation	27
7.1. Mutexes	27
7.2. Barriers	27
7.3. Condvars	27
7.4. Andere Synchronisationstypen	27
Kapitel 8. Sockets	28
8.1. Socket-Optionen	28
Kapitel 9. Gemeinsam verwendeter Speicher	32
Kapitel 10. Gemeinsam verwendete Bibliotheken	33
Teil III. Bibliotheksdienste	34
Kapitel 11. Einstellen des Schedulers	35
11.1. Verwendung von chrt zum Einstellen des Schedulers	35
11.2. Preemption	36
11.3. Verwendung von Bibliotheksaufrufen zur Einstellung der Priorität	36

11.3.1. sched_getscheduler	37
11.3.2. sched_setscheduler	38
11.3.3. sched_getparam und sched_setparam	39
11.3.4. sched_get_priority_min und sched_get_priority_max	39
11.3.5. sched_rr_get_interval	40
Kapitel 12. Erstellen von Threads und Prozessen	42
Kapitel 13. Mmap	43
Kapitel 14. Systemaufrufe	44
14.1. sched_yield	44
14.2. getrusage()	44
Kapitel 15. Timestamping	45
15.1. Hardware-Uhren	45
15.1.1. Ablesen von Hardware-Uhren	46
15.2. POSIX-Uhren	48
15.2.1. CLOCK_MONOTONIC_COARSE und CLOCK_REALTIME_COARSE	49
15.2.2. Verwenden von clock_getres() zum Vergleich der Zeitauflösung	49
15.2.3. Verwenden von C-Code zum Vergleich der Zeitauflösung	50
15.2.4. Verwenden des time-Befehls zum Vergleich des Aufwands für Ableseoperationen	51
Kapitel 16. Weitere Informationen	53
16.1. Melden von Fehlern	53
16.2. Weitere Informationsquellen	53
Versionsgeschichte	55

Vorwort

Red Hat Enterprise MRG

Dieses Buch enthält grundlegende Informationen zur Installation und Optimierung der MRG Realtime Komponente von Red Hat Enterprise MRG. Red Hat Enterprise MRG ist eine Hochleistungsplattform für verteiltes Rechnen, die aus drei Komponenten besteht:

1. **Messaging** — plattformübergreifendes und zuverlässiges Hochleistungs-Messaging unter Verwendung des Advanced Message Queuing Protocol (AMQP) Standards.
2. **Realtime** — Konsistent niedrige Latenz und berechenbare Reaktionszeiten für Applikationen, die Latenz im Bereich von Mikrosekunden erfordern.
3. **Grid** — Verteiltes High Throughput Computing (HTC) und High Performance Computing (HPC).

Alle drei Komponenten von Red Hat Enterprise MRG sind für den Einsatz mit der Plattform konzipiert worden, können jedoch auch separat verwendet werden.

MRG Realtime

Zahlreiche Branchen und Organisationen benötigen extrem hochleistungsfähige Rechenumgebungen sowie eine niedrige und berechenbare Latenz, insbesondere die Finanz- und Telekommunikationsbranche. Latenz, oder Reaktionszeit, ist definiert als die Zeit zwischen einem Ereignis und der Systemreaktion darauf und wird in der Regel in Mikrosekunden (μ s) gemessen. Für die meisten in einer Linux-Umgebung laufenden Applikationen kann eine einfache Leistungsoptimierung die Latenz in ausreichendem Maße verbessern. Für diejenigen Branchen jedoch, in denen Latenz nicht nur gering, sondern auch berechenbar und vorhersehbar sein muss, hat Red Hat jetzt einen 'drop-in' Kernel-Ersatz entwickelt, der diesen Anforderungen gerecht wird. MRG Realtime wird als Teil von Red Hat Enterprise MRG vertrieben und bietet eine nahtlose Integration mit Red Hat Enterprise Linux 6. Mit MRG Realtime haben Kunden die Möglichkeit, Latenzzeiten in ihrer Organisation zu messen, zu konfigurieren und aufzeichnen.

1. Dokumentkonventionen

Dieses Handbuch verwendet mehrere Konventionen, um bestimmte Wörter und Sätze hervorzuheben und Aufmerksamkeit auf bestimmte Informationen zu lenken.

In PDF- und Papirausgaben verwendet dieses Handbuch Schriftbilder des [Liberation-Fonts](#)-Sets. Das Liberation-Fonts-Set wird auch für HTML-Ausgaben verwendet, falls es auf Ihrem System installiert ist. Falls nicht, werden alternative, aber äquivalente Schriftbilder angezeigt. Beachten Sie: Red Hat Enterprise Linux 5 und die nachfolgende Versionen beinhalten das Liberation-Fonts-Set standardmäßig.

1.1. Typografische Konventionen

Es werden vier typografische Konventionen verwendet, um die Aufmerksamkeit auf bestimmte Wörter und Sätze zu lenken. Diese Konventionen und die Umstände, unter denen sie auftreten, sind folgende:

Nichtproportional Fett

Dies wird verwendet, um Systemeingaben hervorzuheben, einschließlich Shell-Befehle, Dateinamen und -pfade. Es wird ebenfalls zum Hervorheben von Tasten und Tastenkombinationen verwendet. Zum Beispiel:

Um den Inhalt der Datei **my_next_bestselling_novel** in Ihrem aktuellen Arbeitsverzeichnis zu sehen, geben Sie den Befehl **cat my_next_bestselling_novel**

in den Shell-Prompt ein und drücken Sie **Enter**, um den Befehl auszuführen.

Das oben aufgeführte Beispiel beinhaltet einen Dateinamen, einen Shell-Befehl und eine Taste. Alle werden nichtproportional fett dargestellt und alle können, dank des Kontextes, leicht unterschieden werden.

Tastenkombinationen unterscheiden sich von einzelnen Tasten durch das Pluszeichen, das die einzelnen Teile einer Tastenkombination miteinander verbindet. Zum Beispiel:

Drücken Sie **Enter**, um den Befehl auszuführen.

Drücken Sie **Strg+Alt+F2**, um zu einem virtuellen Terminal zu wechseln.

Das erste Beispiel hebt die zu drückende Taste hervor. Das zweite Beispiel hebt eine Tastenkombination hervor: eine Gruppe von drei Tasten, die gleichzeitig gedrückt werden müssen.

Falls Quellcode diskutiert wird, werden Klassennamen, Methoden, Funktionen, Variablennamen und Rückgabewerte, die innerhalb eines Abschnitts erwähnt werden, wie oben gezeigt **nichtproportional fett** dargestellt. Zum Beispiel:

Zu dateiverwandten Klassen zählen **filesystem** für Dateisysteme, **file** für Dateien und **dir** für Verzeichnisse. Jede Klasse hat ihren eigenen Satz an Berechtigungen.

Proportional Fett

Dies kennzeichnet Wörter oder Sätze, die auf einem System vorkommen, einschließlich Applikationsnamen, Text in Dialogfeldern, beschriftete Schaltflächen, Bezeichnungen für Auswahlkästchen und Radio-Buttons, Überschriften von Menüs und Untermenüs. Zum Beispiel:

Wählen Sie **System** → **Einstellungen** → **Maus** in der Hauptmenüleiste aus, um die **Mauseinstellungen** zu öffnen. Wählen Sie im Reiter **Tasten** auf das Auswahlkästchen **Mit links bediente Maus** und anschließend auf **Schließen**, um die primäre Maustaste von der linken auf die rechte Seite zu ändern (d.h., um die Maus auf Linkshänder anzupassen).

Um ein Sonderzeichen in eine **gedit**-Datei einzufügen, wählen Sie **Anwendungen** → **Zubehör** → **Zeichentabelle** aus der Hauptmenüleiste. Wählen Sie als Nächstes **Suchen** → **Suchen** aus der Menüleiste der **Zeichentabelle**, geben Sie im Feld **Suchbegriff** den Namen des Zeichens ein und klicken Sie auf **Weitersuchen**. Das gesuchte Zeichen wird daraufhin in der **Zeichentabelle** hervorgehoben. Doppelklicken Sie auf dieses hervorgehobene Zeichen, um es in das Feld **Zu kopierender Text** zu übernehmen und klicken Sie anschließend auf die Schaltfläche **Kopieren**. Gehen Sie nun zurück in Ihr Dokument und wählen Sie **Bearbeiten** → **Einfügen** aus der **gedit**-Menüleiste.

Der oben aufgeführte Text enthält Applikationsnamen, systemweite Menünamen und -elemente, applikationsspezifische Menünamen sowie Schaltflächen und Text innerhalb einer grafischen Oberfläche. Alle werden proportional fett dargestellt und sind anhand des Kontextes unterscheidbar.

Nichtproportional Fett Kursiv oder **Proportional Fett Kursiv**

Sowohl bei nichtproportional fett als auch bei proportional fett weist ein zusätzlicher Kursivdruck auf einen ersetzbaren oder variablen Text hin. Kursivdruck kennzeichnet Text, der nicht wörtlich eingegeben wird, oder angezeigten Text, der sich abhängig von den gegebenen Umständen unterscheiden kann. Zum Beispiel:

Um sich mit einer Remote-Maschine via SSH zu verbinden, geben Sie an einem Shell-

Prompt **ssh *username@domain.name*** ein. Falls die Remote-Maschine **example.com** ist und Ihr Benutzername auf dieser Maschine John lautet, geben Sie also **ssh john@example.com** ein.

Der Befehl **mount -o remount *file-system*** hängt das angegebene Dateisystem wieder ein. Um beispielsweise das **/home**-Dateisystem wieder einzuhängen, verwenden Sie den Befehl **mount -o remount /home**.

Um die Version des derzeit installierten Pakets zu sehen, verwenden Sie den Befehl **rpm -q *package***. Die Ausgabe sieht wie folgt aus: ***package-version-release***.

Beachten Sie die kursiv dargestellten Begriffe oben — *username*, *domain.name*, *file-system*, *package*, *version* und *release*. Jedes Wort ist ein Platzhalter entweder für Text, den Sie für einen Befehl eingeben, oder für Text, der vom System angezeigt wird.

Neben der Standardbenutzung für die Darstellung des Titels eines Werks zeigt der Kursivdruck auch die erstmalige Verwendung eines neuen und wichtigen Begriffs an. Zum Beispiel:

Publican ist ein *DocBook* Publishing-System.

1.2. Konventionen für Seitenansprachen

Ausgaben des Terminals und Auszüge aus dem Quellcode werden visuell vom umliegenden Text hervorgehoben durch sogenannte Seitenansprachen (auch Pull-Quotes genannt).

Eine an das Terminal gesendete Ausgabe wird in den Schrifttyp **nichtproportional Roman** gesetzt und wie folgt dargestellt:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Auszüge aus dem Quellcode werden ebenfalls in den Schrifttyp **nichtproportional Roman** gesetzt, doch wird zusätzlich noch die Syntax hervorgehoben:

```

static int kvm_vm_ioctl_deassign_device(struct kvm *kvm,
                                       struct kvm_assigned_pci_dev *assigned_dev)
{
    int r = 0;
    struct kvm_assigned_dev_kernel *match;

    mutex_lock(&kvm->lock);

    match = kvm_find_assigned_dev(&kvm->arch.assigned_dev_head,
                                assigned_dev->assigned_dev_id);
    if (!match) {
        printk(KERN_INFO "%s: device hasn't been assigned before, "
                    "so cannot be deassigned\n", __func__);
        r = -EINVAL;
        goto out;
    }

    kvm_deassign_device(kvm, match);

    kvm_free_assigned_device(kvm, match);

out:
    mutex_unlock(&kvm->lock);
    return r;
}

```

1.3. Anmerkungen und Warnungen

Zu guter Letzt verwenden wir drei visuelle Stile, um die Aufmerksamkeit auf Informationen zu lenken, die andernfalls vielleicht übersehen werden könnten.



Anmerkung

Eine Anmerkung ist ein Tipp, ein abgekürztes Verfahren oder ein alternativer Ansatz für die vorliegende Aufgabe. Das Ignorieren von Anmerkungen sollte keine negativen Auswirkungen haben, aber Sie verpassen so vielleicht einen Trick, der Ihnen das Leben vereinfachen könnte.



Wichtig

Die Wichtig-Schaukästen lenken die Aufmerksamkeit auf Dinge, die sonst leicht übersehen werden können: Konfigurationsänderungen, die nur für die aktuelle Sitzung gelten oder Dienste, für die ein Neustart nötig ist, bevor eine Aktualisierung wirksam wird. Das Ignorieren von Wichtig-Schaukästen würde keinen Datenverlust verursachen, kann aber unter Umständen zu Ärgernissen und Frustration führen.



Warnung

Eine Warnung sollte nicht ignoriert werden. Das Ignorieren von Warnungen führt mit hoher Wahrscheinlichkeit zu Datenverlust.

2. Hilfe bekommen und Feedback geben

2.1. Brauchen Sie Hilfe?

Falls Sie Schwierigkeiten mit einer der in diesem Handbuch beschriebenen Prozeduren haben, besuchen Sie das Red Hat Kundenportal unter <http://access.redhat.com>. Via Kundenportal können Sie:

- eine Knowledgebase bestehend aus Artikeln rund um technischen Support für Red Hat Produkte durchsuchen oder zu durchstöbern.
- einen Support-Case bei Red Hat Global Support Services (GSS) einreichen.
- auf weitere Produktdokumentationen zugreifen.

Red Hat unterhält außerdem eine Vielzahl von Mailing-Listen zur Diskussion über Red Hat Software und Technologie. Eine Übersicht der öffentlich verfügbaren Listen finden Sie unter <https://www.redhat.com/mailman/listinfo>. Klicken Sie auf den Namen einer Liste für weitere Einzelheiten zum Abonnieren dieser Liste oder um auf deren Archiv zuzugreifen.

2.2. Wir freuen uns auf Ihr Feedback!

Wenn Sie einen Fehler in diesem Handbuch finden oder eine Idee haben, wie dieses verbessert werden könnte, freuen wir uns über Ihr Feedback! Reichen Sie einen Fehlerbericht für die Komponente **Red Hat Enterprise MRG** in Bugzilla unter <http://bugzilla.redhat.com/> ein.

Vergewissern Sie sich beim Einreichen eines Fehlerberichts, dass Sie die Kennung des Handbuchs mit angeben: *Realtime_Reference_Guide*

Falls Sie uns einen Vorschlag zur Verbesserung der Dokumentation senden möchten, sollten Sie hierzu möglichst genaue Angaben machen. Wenn Sie einen Fehler gefunden haben, geben Sie bitte die Nummer des Abschnitts und einen Ausschnitt des Textes an, damit wir diesen leicht finden können.

Teil I. Hardware

Die Auswahl und Konfiguration der richtigen Hardware ist bei der Einrichtung einer Echtzeitumgebung von entscheidender Bedeutung. Hardware beeinflusst die Art und Weise, wie ein System funktioniert. System Management Interrupts, CPU Cache-Design und NUMA-Nutzung können allesamt unterschiedlich gehandhabt werden. Hardware kann von Anbieter zu Anbieter variieren und nicht jede Hardware eignet sich für Echtzeitumgebungen. Für weitere Informationen werfen Sie bitte einen Blick auf die [Liste für MRG Realtime zertifizierter Hardware](#).

Bei der Einrichtung einer MRG Realtime Umgebung ist es wichtig, dass die Applikation derart konzipiert ist, dass sie mit der verfügbaren Hardware gut interagieren kann. Dieser Abschnitt enthält Informationen über die Art und Weise, wie MRG Realtime mit der Hardware interagiert, und worauf geachtet werden sollte.

Kapitel 1. Prozessorkerne

Ein *Prozessorkern* ist eine physische, zentrale Recheneinheit (Central Processing Unit, kurz CPU) in einem Computer. Kerne sind verantwortlich für die Ausführung von Maschinencode. Ein *Socket* ist die Verbindung zwischen dem Prozessor und dem Motherboard des Computers. Das *Socket* ist die Stelle auf dem Motherboard, an der ein Prozessor platziert wird. Ein Einkern-Prozessor ("Single Core") besetzt physisch ein *Socket* und verfügt über einen einzigen Kern. Ein Vierkern-Prozessor ("Quad Core") besetzt physisch ein *Socket* und verfügt über vier Kerne.

Bei der Entwicklung von Echtzeitanwendungen sollte die Anzahl an verfügbaren Kernen berücksichtigt werden. Zudem sollte beachtet werden, wie Caches von den Kernen gemeinsam verwendet werden und wie die Kerne physisch verbunden sind.

Sind mehrere Kerne für die Applikation verfügbar, verwenden Sie *Threads* oder *Prozesse*, um diese zu nutzen. Ist ein Programm ohne Verwendung dieser Konstrukte geschrieben, so läuft es jeweils nur auf einem Prozessor. Mit einer Mehrkern-Plattform können die Vorteile genutzt werden, die die Verwendung verschiedener Kerne für verschiedene Arten von Operationen bietet.

1.1. Caches

Oft müssen die verschiedenen *Threads* einer Applikation den Zugriff auf eine gemeinsam verwendete Ressource synchronisieren, z.B. eine Datenstruktur. Die Leistung kann in diesem Fall durch Kenntnisse des Cache-Layouts des Systems verbessert werden. Das *Tuna-Tool* kann bei der Bestimmung des Cache-Layout helfen. Versuchen Sie, miteinander interagierende *Threads* an bestimmte Kerne zu binden, so dass sie den Cache gemeinsam verwenden. Ein *gemeinsamer Cache* reduziert Speicherfehler, indem gewährleistet wird, dass das Verfahren zum wechselseitigen Ausschluss (*Mutex*, *Condvar* oder ähnliches) und die Datenstruktur selbst denselben Cache verwenden.

1.2. Interconnects

Es ist wichtig, die zwischen den Kernen vorkommenden *Interconnects* (Zwischenverbindungen) zu untersuchen. Je mehr Kerne eine Maschine hat, desto schwieriger und aufwändiger wird es, allen Kernen gleichbleibend schnellen Speicherzugriff zu gewähren. Viele Hardware-Anbieter bieten jetzt ein transparentes Netzwerk von Verbindungen zwischen Kernen und Speicher, bekannt als *NUMA* (Non-Uniform Memory Access) Architektur. Bei *NUMA*-Systemen gestattet es die Kenntnis der Verbindungstopologie, häufig interagierende *Threads* in aneinander grenzende Kerne zu platzieren.

Kapitel 2. Speicherzuweisung

Linux-basierte Betriebssysteme verwenden ein virtuelles Speichersystem. Jede von einer User-Space-Applikation referenzierte Adresse muss in eine physische Adresse übersetzt werden. Dies erfolgt durch eine Kombination von Seitentabellen und Hardware zur Adressübersetzung im zugrunde liegenden Computersystem.

Da der Übersetzungsmechanismus zwischen einem Programm und dem tatsächlichen Speicher untergebracht ist, kann das Betriebssystem Seiten bei Bedarf "stehlen". Dies erfolgt, indem ein zuvor verwendeter Seitentableneintrag als ungültig markiert wird, so dass das Betriebssystem selbst unter normaler Auslastung Seiten von einer Applikation an eine andere geben kann. Dies kann nachteilige Folgen für System haben, die deterministisches Verhalten benötigen. Anweisungen, die normalerweise in einer festgesetzten Zeitspanne ausgeführt werden, können unter Umständen mehr Zeit benötigen, da ein Seitenfehler ausgelöst wurde.

2.1. Demand Paging

Bei Linux durchlaufen alle durch ein Programm generierten Speicheradressen einen Mechanismus zur Adressübersetzung im Prozessor. Die Adressen werden von einer prozessspezifischen virtuellen Adresse in eine physische Speicheradresse konvertiert. Man spricht dann von *virtuellem Speicher*.

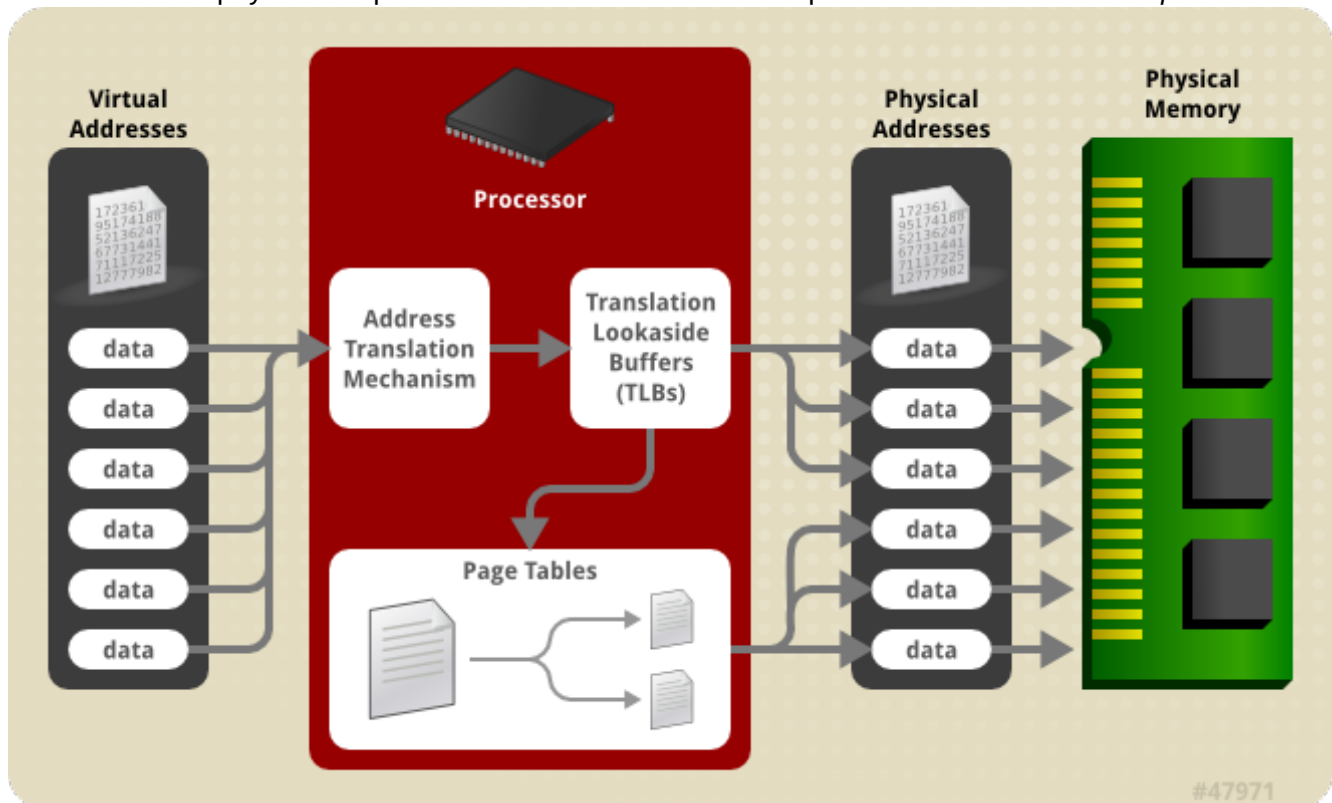


Abbildung 2.1. Virtuelles Speichersystem unter MRG Realtime

Die beiden Hauptkomponenten in diesem Übersetzungsmechanismus sind *Seitentabellen* und *Translation Lookaside Buffers* (Übersetzungspuffer, kurz TLBs). Seitentabellen sind Tabellen auf mehreren Ebenen im physischen Speicher, die Zuordnungen von virtuellem Speicher zu physischem Speicher enthalten. Diese Zuordnungen sind für die Hardware zur Übersetzung virtuellen Speichers im Prozessor lesbar. TLBs sind Caches für Seitentabellenübersetzungen.

Wurde einem Seitentableneintrag eine physische Adresse zugewiesen, wird dieser als *lokale*

Arbeitsmenge ("Working Set") bezeichnet. Muss das Betriebssystem Speicher für andere Prozesse freigeben, kann es Seiten aus der Arbeitsmenge entfernen. Geschieht dies, so kommt es bei einem Verweis auf eine virtuelle Adresse innerhalb dieser Seite zu einem Seitenfehler, und die Seite wird neu zugeordnet. Verfügt das System über extrem wenig physischen Speicher, beginnt das sog. *Thrashing*, wobei der Prozess laufend Seiten von anderen Prozessen stiehlt und keinen Prozess mehr abschließen lässt. Die Statistiken des virtuellen Speichers können durch Überprüfung des ***pgfault***-Werts in der ***/proc/vmstat***-Datei beobachtet werden.

TLBs sind Hardware-Caches von virtuellen Speicherübersetzungen. Jeder Prozessorkern mit einem TLB prüft das TLB parallel zur Initiierung eines Speicher-Lesevorgangs eines Seitentableneintrags. Ist der TLB-Eintrag für eine virtuelle Adresse gültig, so wird der Speicher-Lesevorgang abgebrochen und der Wert im TLB wird für die Adressübersetzung verwendet.

TLBs operieren nach dem Prinzip der *Zugriffslokalität*. Dies bedeutet, wenn ein Code für eine längere Zeitdauer in einer Speicherregion verweilt (wie etwa Schleifen oder Aufruf-bezogene Funktionen), meiden die TLB-Referenzen den Hauptspeicher für die Adressübersetzungen. Dies kann Bearbeitungszeiten wesentlich verkürzen. Beim Schreiben von deterministischem und schnellem Code sollten Sie Funktionen verwenden, die die Zugriffslokalität wahren, z.B. durch Verwendung von Schleifen statt Rekursion. Kann Rekursion nicht vermieden werden, so platzieren Sie den Rekursionsaufruf ans Ende der Funktion. Dies wird als *Endrekursion* bezeichnet und bewirkt, dass der Code in einem relativ kleinen Speicherbereich arbeitet und den Abruf von Tabellenübersetzungen vom Hauptspeicher weitgehend meidet.

Eine potenzielle Quelle von Speicherlatenz ist der *kleine Seitenfehler*. Dazu kommt es, wenn ein Prozess versucht, auf einen Teil des Speichers zuzugreifen, ehe dieser initialisiert wurde. In diesem Fall muss das System einige Operationen durchführen, um die Speicherzuordnungen oder andere Verwaltungsstrukturen zu füllen. Die Schwere eines kleinen Seitenfehlers hängt von der Systemauslastung und anderen Faktoren ab, er hat aber in der Regel nur eine kurze und vernachlässigbare Auswirkung.

Eine schwerere Speicherlatenz wird von einem *großen Seitenfehler* verursacht. Dazu kommt es, wenn das System Speicherpuffer mit der Festplatte synchronisieren muss, Speicherseiten anderer Prozesse auslagern muss oder andere Eingabe/Ausgabe-Aktivitäten zur Freigabe von Speicher durchführen muss. Dies ist der Fall, wenn der Prozessor auf eine Adresse des virtuellen Speichers verweist, der noch keine physische Seite zugewiesen wurde. Beim Verweis auf eine leere Seite löst der Prozessor einen Fehler aus und weist den Kernel-Code an, eine Seite zuzuweisen und zurückzukehren, wodurch die Latenz dramatisch ansteigt.

Beim Schreiben einer Multi-Thread-Applikation ist es wichtig, die Maschinentopologie zu berücksichtigen, wenn man die Daten-Dekomposition entwirft. Die Topologie ist die Speicherhierarchie und umfasst CPU-Caches und den NUMA-Knoten. Die enge gemeinsame Verwendung von Daten auf CPUs in verschiedenen Cache- und NUMA-Domains kann zu Durchsatzproblemen und Engpässen führen.

Konflikte können zu drastischen Leistungsproblemen führen. Bei mancher Hardware unterliegt der Datenverkehr auf den verschiedenen Speicher-Buses keinerlei Regeln zur Vorrangigkeit. Prüfen Sie die verwendete Hardware daher stets, um dies zu vermeiden.

Speicherzuweisungsfehler lassen sich auch mithilfe von CPU-Affinität, Scheduling-Richtlinien und Prioritäten nicht immer vermeiden. Zeigt eine Applikation einen Leistungsabfall, so kann es hilfreich sein zu prüfen, ob dies Auswirkungen von Seitenfehlern sind. Dazu gibt es verschiedene Möglichkeiten, aber eine einfache Methode ist es, sich die Prozessinformationen im ***/proc***-Verzeichnis anzusehen. Nutzen Sie den ***cat***-Befehl zusammen mit der PID des gewünschten Prozesses, um die ***/proc/PID/stat***-Datei einzusehen. Die relevanten Einträge in dieser Datei sind:

► ***Feld 2*** - Dateiname der ausführbaren Datei

- **Feld 10** - Anzahl kleiner Seitenfehler
- **Feld 12** - Anzahl großer Seitenfehler

Trifft ein Prozess auf einen Seitenfehler, so werden alle seine Threads eingefroren, bis der Kernel den Fehler handhabt. Es gibt mehrere Wege, dieses Problem anzugehen, wobei die beste Lösung die Anpassung des Quellcodes zur Vermeidung von Seitenfehlern ist.

Beispiel 2.1. Verwenden der `/proc`-Datei zur Prüfung auf Seitenfehler

Dieses Beispiel nutzt die `/proc`-Datei zur Prüfung auf Seitenfehler in einem laufenden Prozess.

Verwenden Sie den `cat`-Befehl und eine Pipe-Funktion, um nur die zweite, zehnte und zwölfte Zeile der `/proc/PID/stat`-Datei auszugeben:

```
# cat /proc/3366/stat | cut -d\  -f2,10,12
(bash) 5389 0
```

In der Ausgabe oben ist PID 3366 **bash**, und es wurden 5389 kleine Seitenfehler und keine großen Seitenfehler gemeldet.



Anmerkung

Weitere Informationen diesbezüglich finden Sie in dem folgenden Buch:

- *Linux System Programming* von Robert Love

2.2. Die Verwendung von `mlock` zur Vermeidung von Speicherfehlern

Die `mlock`- und `mlockall`-Systemaufrufe teilen dem System mit, dass nur ein bestimmter Speicherbereich verwendet werden darf und dass kein Speicher ausgelagert werden soll. Wurde eine physische Seite also einmal zum Seitentableneintrag zugeordnet, werden Verweise auf diese Seite nicht mehr zu Fehlern führen.

Es gibt zwei Gruppen von `mlock`-Systemaufrufen. Die `mlock`- und `munlock`-Aufrufe sperren und entsperren einen spezifischen Adressbereich. Die `mlockall`- und `munlockall`-Aufrufe sperren oder entsperren den gesamten Programmbereich.

Die Verwendung von `mlock`-Aufrufen sollte genauestens geprüft und mit Vorsicht eingesetzt werden. Handelt es sich um eine große Applikation oder besitzt sie eine große Daten-Domain, können `mlock`-Aufrufe zu Thrashing führen, wenn das System keinen Speicher für andere Aufgaben zuordnen kann. Hat die Applikation mit einem zeitsensiblen Bereich von Code zu tun, kann ein zuvor erfolgter `mlockall`-Aufruf, gefolgt von einem `munlockall`-Aufruf die Auslagerung im kritischen Abschnitt reduzieren. Auch kann `mlock` auf einem Datenbereich angewendet werden, der relativ statisch ist oder nur langsam wächst, auf den jedoch ohne Seitenfehler zugegriffen werden muss.

Die Verwendung von `mlock` ist keine Garantie dafür, dass es nicht zu Seitenfehlern kommt. Es wird verwendet, um sicherzustellen, dass Daten im Speicher bleiben, kann aber nicht sicherstellen, dass diese auf derselben Seite bleiben. Andere Funktionen wie `move_pages` und Speicherverdichter können

Daten trotz **mlock** verschieben.



Wichtig

Verwenden Sie **mlock** stets mit Vorsicht. Die übermäßige Verwendung kann zu einem **out of memory** (OOM) Fehler führen. Platzieren Sie nicht einfach einen **mlockall**-Aufruf an den Start Ihrer Applikation. Es empfiehlt sich, nur die Daten und den Text des Echtzeitabschnitts der Applikation zu sperren.

Beispiel 2.2. Verwendung von mlock in einer Applikation

Dieses Beispiel verwendet den **mlock**-Aufruf in einer einfachen Applikation.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>

char *
alloc_workbuf(int size)
{
    char *ptr;

    /* allocate some memory */
    ptr = malloc(size);

    /* return NULL on failure */
    if (ptr == NULL)
        return NULL;

    /* lock this buffer into RAM */
    if (mlock(ptr, size)) {
        free(ptr);
        return NULL;
    }
    return ptr;
}

void
free_workbuf(char *ptr, int size)
{
    /* unlock the address range */
    munlock(ptr, size);

    /* free the memory */
    free(ptr);
}
```



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- » `mlock(2)`
- » `mlock(3)`
- » `mlockall(2)`
- » `move_pages(2)`

Kapitel 3. Hardware-Interrupts

Hardware-Interrupts werden von Geräten zur Mitteilung darüber verwendet, dass sie die Aufmerksamkeit des Betriebssystems benötigen. Einige gängige Beispiele sind etwa die Festplatte, die signalisiert, dass eine Reihe von Datenblocks gelesen wurde oder dass ein Netzwerkgerät einen Puffer mit Netzwerkpaketen verarbeitet hat. Interrupts werden auch für asynchrone Ereignisse verwendet, so etwa für die Ankunft neuer Daten von einem externen Netzwerk. Hardware-Interrupts werden mittels eines kleinen Netzwerks von Interrupt-Management- und Routing-Geräten direkt an die CPU geschickt. Dieses Kapitel beschreibt die verschiedenen Arten von Interrupts und deren Verarbeitung durch die Hardware und das Betriebssystem. Es geht weiterhin darauf ein, wie der MRG Realtime Kernel sich hinsichtlich der Handhabung der Interrupt-Typen vom Standard-Kernel unterscheidet.

Ein Standardsystem empfängt im Laufe seiner gesamten Betriebszeit mehrere Millionen von Interrupts, darunter ein halbregelmäßiger "Timer"-Interrupt, der periodische Wartungs- und System-Scheduling-Entscheidungen durchführt. Es kann auch spezielle Interrupts empfangen, wie etwa NMI (Non-Maskable Interrupts) und SMI (System Management Interrupts).

Hardware Interrupts werden durch eine *Interrupt-Nummer* referenziert. Diese Nummern sind der jeweiligen Hardware zugeordnet, die den Interrupt auslöst. Dies ermöglicht es dem System zu überwachen, welches Gerät den Interrupt erstellt hat und wann dies stattgefunden hat.

In den meisten Computersystemen werden Interrupts so schnell wie möglich gehandhabt. Bei Erhalt eines Interrupts wird die aktuelle Aktivität gestoppt und ein *Interrupt-Handler* wird ausgeführt. Der Handler unterbricht andere laufende Programme und Systemaktivitäten, was unter Umständen das gesamte System verlangsamt und Latenzen verursacht. MRG Realtime verändert die Art und Weise, in der Interrupts gehandhabt werden, um die Leistung zu verbessern und Latenzen zu verringern.

Beispiel 3.1. Anzeigen von Interrupts in Ihrem System

Um die von einem Linux-System erhaltene Art und Anzahl an Hardware-Interrupts zu untersuchen, verwenden Sie den **cat**-Befehl, um **/proc/interrupts** einzusehen:

```
$ cat /proc/interrupts
CPU0      CPU1
0: 13072311      0      IO-APIC-edge      timer
1: 18351         0      IO-APIC-edge      i8042
8: 190           0      IO-APIC-edge      rtc0
9: 118508        5415    IO-APIC-fasteoi    acpi
12: 747529       86120   IO-APIC-edge      i8042
14: 1163648      0        IO-APIC-edge      ata_piix
15: 0            0        IO-APIC-edge      ata_piix
16: 12681226     126932   IO-APIC-fasteoi    ahci, uhci_hcd:usb2, radeon, yenta,
eth0
17: 3717841      0        IO-APIC-fasteoi    uhci_hcd:usb3, HDA, iwl3945
18: 0            0        IO-APIC-fasteoi    uhci_hcd:usb4
19: 577          68       IO-APIC-fasteoi    ehci_hcd:usb1, uhci_hcd:usb5
NMI: 0           0        Non-maskable interrupts
LOC: 3755270     9388684   Local timer interrupts
RES: 1184857     2497600   Rescheduling interrupts
CAL: 12471       2914     function call interrupts
TLB: 14555       15567    TLB shootdowns
TRM: 0           0        Thermal event interrupts
SPU: 0           0        Spurious interrupts
ERR: 0
MIS: 0
```

Die Ausgabe zeigt die verschiedenen Arten von Hardware-Interrupts, wie viele empfangen wurden, welche CPU Ziel des Interrupts war, sowie das Gerät, das den Interrupt generiert hat.

3.1. Level-Signalled Interrupts

Level-Signalled Interrupts verwenden eine spezielle Interrupt-Leitung zur Übertragung von elektrischen Spannungen.

Diese spezielle Leitung kann eine von zwei Spannungen übertragen, um eine binäre 1 oder binäre 0 darzustellen. Wurde ein Signal über die Leitung gesendet, so bleibt der Status erhalten, bis die CPU diesen ausdrücklich zurücksetzt. Dies erfolgt, indem die CPU das generierende Gerät dazu auffordert, die Leitung wieder freizugeben. Dies ermöglicht es einer Reihe von Geräten, sich eine einzige Interrupt-Leitung zu teilen. Hat die CPU ein Gerät angewiesen, die Leitung freizugeben, die Leitung bleibt jedoch aktiv, so bedeutet das, dass ein weiterer Interrupt aussteht.

Obwohl Level-Signalled Interrupts ein hohes Maß an Hardware-Logik sowohl auf den Geräten als auch auf der CPU erfordern, bieten sie auch eine Reihe von Vorteilen: Sie können von mehr als nur einem Gerät verwendet werden und verpassen so gut wie keine Interrupts.

3.2. Message-Signalled Interrupts

Viele moderne Systeme verwenden *Message-Signalled Interrupts*, die die Signale als spezielle Nachricht über einen Paket- oder Nachrichten-basierten elektrischen Bus versenden.

Ein gängiges Beispiel für diesen Bus-Typ ist PCI Express (Peripheral Component Interconnect Express)

oder PCIe). Diese Geräte übermitteln eine Nachricht als einen Typ, den der PCIe-Host-Controller als Interrupt-Nachricht interpretiert. Der Host-Controller schickt die Nachricht dann weiter an die CPU.

Je nach Hardware kann ein PCIe-System das Signal mittels spezieller Interrupt-Leitung zwischen PCIe Host-Controller und der CPU senden, oder es kann die Nachricht z.B. über den CPU HyperTransport Bus verschicken. Viele PCIe-Systeme können auch im Legacy-Modus laufen, bei dem traditionelle Interrupt-Leitungen implementiert sind, um ältere Betriebssysteme zu unterstützen, oder Linux-Kernels mit der Option ***pci=noms*** an der Kernel-Befehlszeile gebootet werden.

3.3. Non-maskable Interrupts

Ein Interrupt wird als *maskiert* bezeichnet, wenn er deaktiviert wurde oder wenn die CPU angewiesen wurde, ihn zu ignorieren. Ein *Non-Maskable Interrupt* (nicht-maskierbarer Interrupt, kurz NMI) kann nicht ignoriert werden und wird in der Regel nur für kritische Hardware-Fehler verwendet.

NMIs werden normalerweise über eine separate Interrupt-Leitung gesendet. Wird ein NMI von der CPU empfangen, so zeigt dies an, dass es zu einem kritischen Fehler gekommen ist und das System wahrscheinlich abstürzen wird. Der NMI ist in der Regel die beste Indikation darüber, was das Problem verursacht haben könnte.

Da NMIs nicht ignoriert werden können, werden sie von von einigen Systemen als Hardware-Monitor verwendet. Das Gerät sendet eine Folge von NMIs, die von einem NMI-Handler im Prozessor geprüft werden. Sind bestimmte Bedingungen erfüllt - wie etwa das Nichtauslösen eines Interrupts nach einer bestimmten Zeitdauer - kann der NMI-Handler eine Warnung und Informationen zur Fehlerbehebung generieren. Dies hilft bei der Identifikation und Vorbeugung von System-Lockups.

3.4. System Management Interrupts

System Management Interrupts (SMIs) werden für erweiterte Funktionalität verwendet, wie etwa zur Emulation veralteter Hardware-Geräte. Sie können auch für Systemverwaltungsaufgaben eingesetzt werden. SMIs ähneln NMIs insofern, als sie ebenfalls eine spezielle elektrische Leitung direkt in die CPU verwenden und in der Regel nicht maskiert werden können.

Wird ein SMI empfangen, so startet die CPU den *System Management Mode* (SMM). In diesem Modus wird eine Handler-Routine auf niedriger Ebene ausgeführt, um die SMIs zu handhaben. Der SMM wird in der Regel direkt von der Systemverwaltungs-Firmware - oft dem BIOS oder EFI - bereitgestellt.

SMIs werden am häufigsten zur Emulation veralteter Hardware verwendet. Ein gängiges Beispiel ist die Emulation eines Floppy-Laufwerks. Ist kein Floppy-Laufwerk am System angeschlossen, so kann stattdessen eine virtualisierte, netzwerkverwaltete Emulation verwendet werden. Versucht das Betriebssystem auf das Floppy-Laufwerk zuzugreifen, so wird ein SMI ausgelöst und der Handler liefert dem Betriebssystem stattdessen ein emuliertes Gerät. Das Betriebssystem behandelt die Emulation dann wie das echte, veraltete Gerät.

SMIs können sich nachteilig auf MRG Realtime auswirken, weil diese ohne direkte Beteiligung des Betriebssystems stattfinden. Eine schlecht geschriebene SMI-Handling-Routine kann viele Millisekunden an CPU-Zeit verbrauchen und das Betriebssystem kann den Handler nicht unterbrechen, falls dies nötig ist. Diese Situation führt zu periodisch hohen Latenzen in einem ansonsten gut optimierten, schnell reagierenden System. Da SMI-Handler leider von Herstellern zur Steuerung der CPU-Temperatur und Ventilation verwendet werden können, ist es nicht möglich, sie zu deaktivieren. Stattdessen sollten Sie den Hersteller auf das Problem aufmerksam machen.



Anmerkung

Sie können versuchen, SMIs auf einem MRG Realtime System mithilfe des **hwlatdetect**-Hilfsprogramms zu isolieren. Dieses Hilfsprogramm steht im **rt-tests**-Paket zur Verfügung und wurde dazu entwickelt, die Zeit zu messen, die eine CPU von einer SMI-Handling-Routine in Beschlag genommen wird.

3.5. Advanced Programmable Interrupt Controller

Der *Advanced Programmable Interrupt Controller* (APIC) wurde von Intel® entwickelt, um eine hohe Anzahl von Interrupts verwalten zu können (so dass diese je nach Bedarf befehlsorientiert an einen bestimmten Satz verfügbarer CPUs geroutet werden können), um die Kommunikation zwischen CPUs zu unterstützen und um zu vermeiden, dass sich eine hohe Anzahl an Geräten eine einzelne Interrupt-Leitung teilen muss.

APIC steht für eine Reihe von Geräten und Technologien, die zusammenarbeiten, um eine hohe Anzahl an Hardware-Interrupts zu generieren, routen und handzuhaben. Es verwendet eine Kombination aus einem lokalen APIC, der in jede System-CPU eingebaut ist, sowie eine Reihe von Eingabe/Ausgabe-APICs, die direkt mit Hardware-Geräten verbunden sind. Generiert ein Hardware-Gerät einen Interrupt, so wird dies vom mit ihm verbundenen I/O-APIC aufgespürt und dann über den System-APIC-Bus an eine bestimmte CPU geroutet. Das Betriebssystem weiß aus einer Kombination von Informationsquellen, welcher I/O-APIC mit welchem Gerät und mit welcher bestimmten Interrupt-Leitung innerhalb dieses Geräts verbunden ist. Zunächst ist da die ACPI DSDT (Advanced Configuration and Power Interface Differentiated System Description Table), die Information zur spezifischen Verkabelung des Host-System Motherboards und Peripherie-Komponenten beinhaltet. Des Weiteren liefert ein Gerät bestimmte Informationen zu seinen verfügbaren Interrupt-Quellen. Gemeinsam bieten diese beiden Datensätze Informationen zur allgemeinen Interrupt-Hierarchie.

Möglich sind komplexe APIC-basierte Strategien zur Interrupt-Verwaltung, bei denen die System-APICs in Hierarchien verbunden sind und Interrupts in lastverteilter Weise an CPUs geliefert werden, statt eine bestimmte CPU oder einen Satz an CPUs zum Ziel zu haben.

Teil II. Applikationsarchitektur

Der MRG Realtime Kernel bietet eine Reihe von Konstrukten, die Softwareentwicklern dabei helfen sollen, Applikationen mit den höchst möglichen Leistungsstandards zu entwickeln. Dieser Abschnitt behandelt diese Features und deren Verwendung.

In diesem und den nächsten Abschnitten werden Anleitungen dazu geliefert, wie der MRG Realtime Kernel direkt optimiert werden kann. Die meisten Änderungen können auch mithilfe eines Tools namens *Tuna* durchgeführt werden. Es verfügt über eine grafische Oberfläche oder kann über die Befehlszeile ausgeführt werden.

Tuna kann zur Änderung von Attributen von Threads und Interrupts, etwa Scheduling-Richtlinie, Scheduler-Priorität und Prozessoraffinität verwendet werden. Das Tool wurde für die Verwendung auf einem laufenden System konzipiert und Änderungen werden sofort wirksam. Auf diese Weise kann jedes applikationsspezifische Mess-Tool die Systemleistung sofort nach den vorgenommenen Änderungen sehen und analysieren.

Informationen zur Installation und Verwendung von Tuna finden Sie im *MRG Tuna Benutzerhandbuch*.

Kapitel 4. Threads und Prozesse

Zwar verwenden alle Programme Threads und Prozesse, MRG Realtime handhabt diese jedoch auf andere Weise als standardmäßige Red Hat Enterprise Linux Systeme. Dieses Kapitel erläutert die MRG Realtime Handhabung von Threads und Prozessen.

Jeder CPU-Kern unterliegt Beschränkungen hinsichtlich der Menge an Arbeit, die er durchführen kann. Um eine größere Effizienz zu erzielen, können Applikationen verschiedene Aufgaben simultan auf mehreren Kernen ausführen. Man spricht dann von *Parallelisierung*.

Programme können mithilfe von *Threads* parallelisiert werden. Allerdings werden Threads und Prozesse oft miteinander verwechselt, daher ist es wichtig, die Unterschiede dieser Begriffe zu verstehen.

Prozess

Ein UNIX®-artiger Prozess ist ein Betriebssystemskonstrukt, das Folgendes enthält:

1. Adresszuordnungen für virtuellen Speicher
2. Einen Ausführungskontext (PC, Stack, Registers)
3. Status-/Accounting-Informationen

Linux-Prozesse begannen als genau diese Art von Prozess. Als das Konzept von mehr als einem Prozess in einem Adressbereich entwickelt wurde, wendete sich Linux einer Prozessstruktur zu, die einen Adressbereich mit anderen Prozessen teilt. Dies funktioniert gut, so lange die Prozessdatenstruktur gering gehalten wird. Für den Rest dieses Dokuments gilt, dass der Begriff *Prozess* sich auf einen unabhängigen Adressbereich bezieht, der möglicherweise mehrere Threads enthält.

Thread

Streng genommen handelt es sich bei einem Thread um eine terminierbare Einheit, die Folgendes enthält:

1. Einen Programmzähler (PC)
2. Einen Register-Kontext
3. Einen Stapelzeiger

Innerhalb eines Prozesses können mehrere Threads existieren.

Beim Programmieren auf einem MRG Realtime System gibt es zwei Möglichkeiten, Programme zu parallelisieren:

1. Verwenden der **fork**- und **exec**-Funktionen zur Erstellung neuer Prozesse
2. Verwenden der Posix-Threads (pthreads) Programmierschnittstelle zur Erstellung neuer Threads innerhalb eines bereits laufenden Prozesses



Anmerkung

Prüfen Sie, wie die Komponenten untereinander interagieren, ehe Sie entscheiden, wie diese parallelisiert werden sollen. Sind die Komponenten unabhängig voneinander und werden nur wenig oder gar nicht interagieren, dann ist die Erstellung eines neuen Adressbereichs und die Ausführung als neuer Prozess in der Regel die bessere Option. Falls sich die Komponenten jedoch Daten teilen oder häufig kommunizieren, so ist deren Ausführung als Threads innerhalb eines Adressbereichs in der Regel effizienter.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten bzw. in den folgenden Büchern:

- » `fork(2)`
- » `exec(2)`
- » *Programming with POSIX Threads*, David R. Butenhof, Addison-Wesley, ISBN 0-201-63392-2
- » *Advanced Programming in the UNIX Environment*, Zweite Ausgabe, W. Richard Stevens und Stephen A. Rago, Addison-Wesley, ISBN 0-201-43307-9
- » „*POSIX Threads Programming*“, Blaise Barney, Lawrence Livermore National Laboratory, <http://www.llnl.gov/computing/tutorials/pthreads/>

Kapitel 5. Prioritäten und Richtlinien

Alle Linux-Threads besitzen eine der folgenden *Scheduling-Richtlinien*:

- **SCHED_OTHER** oder **SCHED_NORMAL**: Die Standardrichtlinie
- **SCHED_BATCH**: Ähnlich **SCHED_OTHER**, jedoch mit Durchsatz-Fokus
- **SCHED_IDLE**: Eine niedrigere Priorität als **SCHED_OTHER**
- **SCHED_FIFO**: Eine First-in/First-out Echtzeitrichtlinie
- **SCHED_RR**: Eine Round-Robin Echtzeitrichtlinie

Die für MRG Realtime entscheidenden Richtlinien sind **SCHED_OTHER**, **SCHED_FIFO** und **SCHED_RR**.

Die standardmäßige Scheduling-Richtlinie für Linux-Threads ist **SCHED_OTHER** oder **SCHED_NORMAL**. Beide besitzen eine dynamische Priorität, die - basierend auf den Charakteristika des Threads - geändert wird. Ein weiterer Faktor, der die Priorität von **SCHED_OTHER**-Threads beeinflusst, ist deren *Nice-Wert*. Der Nice-Wert ist eine Zahl zwischen -20 (höchste Priorität) und 19 (niedrigste Priorität). Standardmäßig besitzen **SCHED_OTHER**-Threads einen Nice-Wert von 0. Die Anpassung des Nice-Werts ändert die Art und Weise, wie der Thread gehandhabt wird.

Threads mit einer **SCHED_FIFO**-Richtlinie laufen vor **SCHED_OTHER**-Aufgaben. Statt Nice-Werten verwendet **SCHED_FIFO** eine feste Priorität zwischen 1 (am niedrigsten) und 99 (am höchsten). Ein **SCHED_FIFO**-Thread mit einer Priorität 1 wird immer vor einem **SCHED_OTHER**-Thread eingereiht.

Die **SCHED_RR**-Richtlinie ist der **SCHED_FIFO**-Richtlinie sehr ähnlich. Bei der **SCHED_RR**-Richtlinie werden Threads gleicher Priorität nach *Round-Robin*-Verfahren (also reihum) eingeplant. In der Regel wird **SCHED_FIFO** der Vorzug vor **SCHED_RR** gegeben.

SCHED_FIFO- und **SCHED_RR**-Threads werden ausgeführt, bis eines der folgenden Ereignisse eintritt:

- Der Thread pausiert oder beginnt, auf ein Ereignis zu warten
- Ein Echtzeit-Thread mit höherer Priorität ist bereit zu laufen

Findet keines dieser Ereignisse statt, so laufen die Threads auf unbestimmte Dauer auf diesem Prozessor und Threads mit niedrigerer Priorität haben keine Chance zu laufen. Dies kann dazu führen, dass Systemdienst-Threads nicht laufen und Operationen wie Speicherauslagerung und Dateisystem-Flushing nicht wie erwartet erfolgen.

Tabelle 5.1. Richtlinienprioritäten

Richtlinie	Standardmäßiger Prioritätswert	Niedrigster Prioritätswert	Höchster Prioritätswert
SCHED_FIFO		1	99
SCHED_RR		1	99
SCHED_OTHER	0	-20	19

Kapitel 6. Affinität

Jeder Thread und jede Interrupt-Quelle im System besitzt einen *Prozessoraffinität*-Eigenschaftswert. Der Betriebssystem-Scheduler verwendet diese Information, um festzustellen, welche Threads und Interrupts auf welcher CPU laufen sollen.

Die Einstellung der Prozessoraffinität zusammen mit wirksamer Richtlinien- und Prioritäteneinstellung kann zur bestmöglichen Leistung verhelfen. Applikationen befinden sich stets in Konkurrenz mit anderen Prozessen um Ressourcen, insbesondere CPU-Zeit. Je nach Applikation laufen verwandte Threads oftmals auf demselben Kern. Alternativ kann ein Applikations-Thread einem Kern zugeordnet sein.

Systeme, die Multitasking durchführen, sind anfälliger für Indeterminismus. Es ist möglich, dass sogar hoch priorisierte Applikationen zum Warten gezwungen werden, während sich eine niedriger priorisierte Applikation einem kritischen Abschnitt befindet. Nachdem die niedrig priorisierte Applikation den kritischen Abschnitt abgeschlossen hat, kann der Kernel diese sicher unterbrechen und stattdessen die hoch priorisierte Applikation zur Ausführung auf dem Prozessor einplanen. Zudem kann die Migration von Prozessen von einer CPU zur anderen aufgrund von Cache-Invalidierung recht ressourcenintensiv sein. MRG Realtime beinhaltet Tools, die einige dieser Probleme beheben und eine bessere Kontrolle von Latenzen ermöglichen.

Affinität wird als *Bitmaske* dargestellt, wobei jedes Bit in der Maske für einen CPU-Kern steht. Ist das Bit auf 1 gesetzt, dann kann der Thread oder Interrupt auf diesem Kern laufen; ist es auf 0 gesetzt, darf der Thread oder Interrupt nicht auf dem Kern laufen. Standardmäßig ist die Affinitäts-Bitmaske auf Einsen gesetzt, was bedeutet, dass der Thread oder Interrupt auf jedem Kern des Systems laufen darf.

Standardmäßig können Prozesse auf jeder CPU laufen. Allerdings können Prozesse angewiesen werden, auf einer vorab bestimmten Auswahl an CPUs zu laufen, indem die Affinität des Prozesses geändert wird. Untergeordnete Prozesse erben die CPU-Affinität der ihnen übergeordneten Prozesse.

Sehen Sie nachfolgend einige der üblichen Affinitätseinstellungen:

- Ein CPU-Kern wird für alle Systemprozesse reserviert und der Applikation wird gestattet, auf dem Rest der Kerne zu laufen, mit einem CPU-Kern pro Applikations-Thread.
- Ein Applikations-Thread und ein einziger Kernel-Thread (wie etwa Netzwerk-Softirq oder einen Treiber-Thread) wird auf derselben CPU gestattet.
- Erzeuger- und Verbraucher-Threads werden auf jeder CPU gruppiert.

Es wird empfohlen, Affinitätseinstellungen gemeinsam mit dem Programm zu entwerfen, um sie besser auf das erwartete Verhalten abzustimmen.

Zur Optimierung von Affinitäten auf einem Echtzeitsystem wird üblicherweise bestimmt, wie viele Kerne zur Ausführung der Applikation nötig sind, um diese Kerne anschließend zu isolieren. Dies kann entweder mithilfe des **Tuna**-Tools oder durch Verwendung von Shell-Skripten zur Bearbeitung des Bitmasken-Werts erfolgen. Der **taskset**-Befehl kann zur Änderung der Affinität eines Prozesses verwendet werden, während die Bearbeitung des **/proc**-Dateisystemeintrags die Affinität eines Interrupts ändert.

**Anmerkung**

Weitere Informationen diesbezüglich finden Sie in folgendem Handbuch bzw. auf der folgenden Handbuchseite:

- *MRG Tuna Benutzerhandbuch*
- `taskset(1)`

6.1. Verwendung des `taskset`-Befehls zur Einstellung der Prozessoraffinität

Der **`taskset`**-Befehl setzt und prüft Affinitätsinformationen für einen bestimmten Prozess. Das Tuna-Tool kann zu diesem Zweck ebenfalls verwendet werden.

Verwenden Sie den "`taskset`"-Befehl mit der **`-p`**- oder **`--pid`**-Option und der PID des zu prüfenden Prozesses. **`-c`** oder **`--cpu-list`** zeigen die Informationen als numerische Liste von Kernen statt als Bitmaske an.

Der folgende Befehl prüft die Affinität des Prozesses mit PID 1000. In diesem Fall darf PID 1000 entweder CPU 0 oder CPU 1 verwenden:

```
# taskset -p -c 1000
pid 1000's current affinity list: 0,1
```

Die Affinität kann eingestellt werden, indem die Nummer der CPU angegeben wird, an die der Prozess gebunden werden soll. In diesem Beispiel konnte PID 1000 zuvor auf CPU 0 oder CPU 1 laufen, doch die Affinität wurde nun geändert, so dass sie nur noch auf CPU 1 laufen kann:

```
# taskset -p -c 1 1000
pid 1000's current affinity list: 0,1
pid 1000's new affinity list: 1
```

Um mehr als eine CPU-Affinität zu definieren, geben Sie beide CPU-Nummern an, getrennt durch ein Komma:

```
# taskset -p -c 0,1 1000
pid 1000's current affinity list: 1
pid 1000's new affinity list: 0,1
```

Der **`taskset`**-Befehl kann auch zum Starten eines neuen Prozesses mit einer bestimmten Affinität verwendet werden. Dieser Befehl lässt die **`/bin/my-app`**-Applikation auf CPU 4 laufen:

```
# taskset -c 4 /bin/my-app
```

Für eine feinere Kontrolle können auch Priorität und Richtlinie eingestellt werden. Dieser Befehl führt die **`/bin/my-app`**-Applikation auf CPU 4 aus, mit einer **`SCHED_FIFO`**-Richtlinie und einer Priorität von 78:

```
# taskset -c 5 chrt -f 78 /bin/my-app
```

6.2. Verwendung des `sched_getaffinity()`-Systemaufrufs zur Einstellung der Prozessoraffinität

Zusätzlich zum **taskset**-Befehl kann Prozessoraffinität auch mittels des **sched_getaffinity()**-Systemaufrufs eingestellt werden.

Der folgende Codeausschnitt ruft die Informationen zur CPU-Affinität für eine bestimmte PID ab. Wird die PID 0 übergeben, so werden die Informationen zur Affinität des aktuellen Prozesses wiedergegeben:

```
int sched_getaffinity(pid_t pid, size_t setsize, const cpu_set_t *set)
```



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- » `sched_getaffinity(2)`
- » `sched_setaffinity(2)`

Kapitel 7. Thread-Synchronisation

Wenn Threads Zugriff auf gemeinsam verwendete Ressourcen benötigen, so wird dies mithilfe der *Thread-Synchronisation* koordiniert. Die drei Mechanismen zur Thread-Synchronisation bei Linux sind:

1. Mutexes
2. Barriers
3. Condvars

7.1. Mutexes

Das Wort *Mutex* leitet sich vom englischen Begriff *mutual exclusion* (wechselseitiger Ausschluss) ab. Ein Mutex ist ein POSIX-Thread-Konstrukt und wird mittels `pthread_create_mutex`-Bibliothekaufruf erstellt. Ein Mutex serialisiert Zugriff auf jeden Abschnitt von Code, so dass zu jeder Zeit nur ein Thread einer Applikation den Code ausführt.

Ähnlich dem Mutex-Verfahren ist das *Futex*-Verfahren, oder Fast User *muTEX*, bei der es sich um einen internen Mechanismus zur Implementierung von Mutexes handelt. Futexes verwenden gemeinsame Konventionen von Kernel und der C-Bibliothek. Dies gestattet die Sperrung oder Freigabe eines Mutexes im einfachsten Fall ohne Kontextwechsel zum Kernel-Space.

7.2. Barriers

Barriers operieren auf gänzlich andere Weise als andere Thread-Synchronisationsmethoden. Statt den Zugriff auf Coderegionen zu serialisieren, sperren Barriers alle Threads, bis sich eine vorab festgelegte Anzahl an Threads angesammelt hat. Die Barrier gestattet dann die Fortführung aller Threads. Barriers kommen zum Einsatz, wenn eine laufende Applikation sicher sein muss, dass alle Threads ihre Aufgaben erfüllt haben, ehe die Ausführung fortgesetzt werden kann.

7.3. Condvars

Eine *Condvar* oder "Condition Variable" (Bedingungsvariable) ist ein POSIX-Thread-Konstrukt, das auf Erfüllung einer bestimmten Bedingung wartet, ehe fortgefahren wird. In der Regel bezieht sich die signalisierte Bedingung auf den Zustand von Daten, die der Thread sich mit einem anderen Thread teilt. Eine Condvar kann zum Beispiel verwendet werden, um zu signalisieren, dass ein Dateneintrag in eine Bearbeitungwarteschlange platziert wurde und ein Thread, der auf Daten in dieser Warteschlange wartet, jetzt mit der Bearbeitung derselben fortfahren kann.

7.4. Andere Synchronisationstypen

Vor der Einführung von POSIX-Threads wurde Thread-Synchronisation zwischen Prozessen durchgeführt. Die gängigsten Mechanismen waren die System V IPC Aufrufe an gemeinsam verwendeten Speicher, Nachrichtenwarteschlangen und Semaphores. Die Verwendung von System V IPC Aufrufen gilt jetzt als veraltet, stattdessen werden POSIX-Thread-Aufrufe verwendet.

Kapitel 8. Sockets

Ein *Socket* ist ein bidirektionaler Datentransfermechanismus. Sie werden zur Übertragung von Daten zwischen zwei Prozessen verwendet. Die beiden Prozesse können am selben System als Unix-Domain oder Loopback-Sockets laufen oder aber auf verschiedenen Systemen als Netzwerk-Sockets.

Es existieren keine speziellen Optionen oder Einschränkungen zur Verwendung von Sockets auf einem MRG Realtime System.

8.1. Socket-Optionen

Zwei Socket-Optionen sind relevant für MRG Realtime Applikationen: **TCP_NODELAY** und **TCP_CORK**.

TCP_NODELAY

TCP ist das gängigste Transportprotokoll und wird daher in den verschiedensten Anwendungsfällen eingesetzt. Mit der Entwicklung neuer Applikations- und Hardware-Features und der Optimierung der Kernel-Architektur musste TCP neue Heuristiken einführen, um die Änderungen effektiv zu handhaben.

Diese Heuristiken können dazu führen, dass ein Programm instabil wird. Da sich das Verhalten gleichzeitig mit den zugrunde liegenden Betriebssystemkomponenten ändert, sollte man hiermit vorsichtig umgehen.

Ein Beispiel für heuristisches Verhalten in TCP ist die Verzögerung kleiner Speicherpuffer. Dies ermöglicht deren Versendung als ein Netzwerkpaket. Dies funktioniert in der Regel gut, kann aber zu Latenzen führen. Für MRG Realtime Applikationen ist **TCP_NODELAY** eine Socket-Option, die zur Abschaltung dieses Verhaltens verwendet werden kann. Es kann mittels der **setsockopt**-Socket-API mit folgender Funktion aktiviert werden:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

Damit diese Option effektiv ist, müssen die Applikationen kleinere Schreibvorgänge in den Puffer vermeiden, da TCP diese Puffer als individuelle Pakete verschickt. **TCP_NODELAY** kann auch mit anderen Optimierungsheuristiken interagieren, was zu einer insgesamt schlechteren Leistung führen kann.

Besitzen Applikationen mehrere logisch verbundene Puffer, so sollten sie als ein zusammenhängendes Paket verschickt werden, um bessere Latenz und Leistung zu erreichen. Das Paket kann dann komplett mittels eines Sockets mit aktiviertem **TCP_NODELAY** verschickt werden.

Alternativ kann man - wenn die Speicherpuffer zwar logisch verwandt, aber nicht zusammenhängend sind - diese zum Erstellen eines Ein-/Ausgabe-Vektors verwenden. Dieser kann mittels **writv** auf einem Socket mit aktiviertem **TCP_NODELAY** an den Kernel weitergegeben werden.

TCP_CORK

Eine weitere TCP Socket-Option, die auf ähnliche Weise funktioniert, ist **TCP_CORK**. Falls aktiviert, verzögert TCP alle Pakete, bis die Applikation die Sperre entfernt und dadurch die Versendung der zwischengespeicherten Pakete gestattet. Dies ermöglicht Applikationen die Generierung eines Pakets im Kernel-Space, was von Nutzen ist, wenn verschiedene Bibliotheken für Schichtenabstraktionen verwendet werden.

Die **TCP_CORK**-Option kann mittels folgender Funktion aktiviert werden:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

Bei der Aktivierung von **TCP_CORK** spricht man manchmal auch davon, den **Socket zu verkorken**.

In einer Situation, in der der Kernel nicht weiß, wann der metaphorische "Korken" (also die **TCP_CORK**-Option) entfernt werden soll, kann er mittels folgender Funktion entfernt werden:

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

Sobald der Socket "entkorkt" ist, versendet TCP das akkumulierte logische Paket sofort, ohne dass auf weitere Pakete der Applikation gewartet wird.

Beispiel 8.1. Verwendung von **TCP_NODELAY** und **TCP_CORK**

Dieses Beispiel demonstriert die Auswirkungen auf die Leistung, die **TCP_NODELAY** und **TCP_CORK** auf eine Applikation haben können.

Der Server wartet auf Pakete von 30 Bytes und schickt dann ein 2 Byte großes Paket als Antwort. Definieren Sie für den Anfang den TCP-Port und die Anzahl an Paketen, die verarbeitet werden soll. In diesem Beispiel sind es 10.000 Pakete:

```
$ ./tcp_nodelay_server 5001 10000
```

Auf dem Server müssen keine Socket-Optionen eingestellt sein.

Läuft der Client ohne jegliche Parameter, so werden standardmäßige Socket-Optionen verwendet. Verwenden Sie die **no_delay**-Option zur Aktivierung der **TCP_NODELAY** Socket-Optionen. Verwenden Sie die **cork**-Option zur Aktivierung des **TCP_CORK**. In allen Fällen sendet es 15 Pakete von jeweils zwei Bytes und wartet auf Antwort vom Server.

Dieses Beispiel verwendet eine Loopback-Schnittstelle, um drei Varianten zu veranschaulichen.

In der ersten Variante sind weder **TCP_NODELAY** noch **TCP_CORK** in Gebrauch. Dies ist eine Messung der Ausgangswerte. TCP legt Schreibvorgänge zusammen und muss warten, um zu prüfen, ob die Applikation über mehr Daten verfügt als optimal ins Netzwerkpaket passen:

```
$ ./tcp_nodelay_client localhost 5001 10000
10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
```

Die zweite Variante verwendet nur **TCP_NODELAY**. TCP wird dazu angewiesen, keine kleinen Pakete zu zusammenzulegen, sondern Puffer sofort zu verschicken. Dies verbessert die Leistung maßgeblich, erstellt aber für jedes logische Paket eine große Anzahl an Netzwerkpaketen.

```
$ ./tcp_nodelay_client localhost 5001 10000 no_delay
10000 packets of 30 bytes sent in 1649.771240 ms: 181.843399 bytes/ms using
TCP_NODELAY
```

Die dritte Variante verwendet nur **TCP_CORK**. Es halbiert die erforderliche Zeit, um dieselbe Anzahl logischer Pakete zu verschicken, da TCP komplette logische Pakete in seinen Puffern zusammenlegt und insgesamt weniger Netzwerkpakete verschickt.

```
$ ./tcp_nodelay_client localhost 5001 10000 cork
10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using
TCP_CORK
```

Bei diesem Szenario ist **TCP_CORK** das beste Verfahren. Es gestattet der Applikation, exakt mitzuteilen, dass ein Paket fertig ist und ohne Verzögerung verschickt werden muss. Bei der Entwicklung von Programmen, die große Mengen an Daten von einer Datei verschicken müssen, sollten Sie die Verwendung von **TCP_CORK** mit **sendfile** in Erwägung ziehen.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf der folgenden Handbuchseite und in folgenden Beispielapplikationen:

- `sendfile(2)`
- „*TCP Nagle Beispielapplikationen*“, Beispielapplikationen von Ghost-Protokollen geschrieben in C. Sie können diese Applikationen durch Rechtsklick von den folgenden Links herunterladen:
 - http://oops.ghostprotocols.net:81/acme/tcp_nodelay_client.c
 - http://oops.ghostprotocols.net:81/acme/tcp_nodelay_server.c

Kapitel 9. Gemeinsam verwendeter Speicher

Einer der Hauptvorteile von Programm-Threads besteht darin, dass alle in einem Prozesskontext erstellten Threads denselben Adressbereich verwenden. Dies bedeutet, dass sie auf alle Datenstrukturen zugreifen können. Allerdings ist es nicht immer ratsam, dass Applikationen Threads verwenden, die Adressbereiche gemeinsam verwenden; Beispiele dafür sind ein verteiltes Entwicklungsteam oder die Größe der entwickelten Applikation. In diesem Fall müssen separate Prozesse einen Teil des Adressbereichs gemeinsam verwenden. Dies kann auf sowohl Red Hat Enterprise Linux Kernels als auch auf MRG Realtime Linux Kernels mittels *Shared Memory* (gemeinsam verwendetem Speicher) erfolgen.

Der ursprüngliche Mechanismus zur gemeinsamen Verwendung eines Speicherbereichs durch zwei Prozesse war der System V IPC **shmem**-Satz an Aufrufen. Diese Aufrufe sind zwar leistungsfähig, aber recht kompliziert und in der überwiegenden Mehrheit der Anwendungsfälle zu mühselig. Aus diesem Grund sind sie im MRG Realtime Kernel nicht länger enthalten und sollten nicht mehr verwendet werden.

MRG Realtime verwendet POSIX-Aufrufe an gemeinsam verwendeten Speicher wie **shm_open** und **mmap**.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- » `shm_open(3)`
- » `shm_overview(7)`
- » `mmap(2)`

Kapitel 10. Gemeinsam verwendete Bibliotheken

Dynamic Shared Objects (DSOs) werden als *gemeinsam verwendete Bibliotheken* bezeichnet und werden dazu verwendet, um Code zwischen separaten Prozess-Adressbereichen gemeinsam zu verwenden. Die DSO werden einmalig vom **ld.so**-Systemlader geladen. Von dort werden sie dem Adressbereich von Prozessen zugeordnet, die Symbole aus der Bibliothek benötigen. Ehe nicht die erste Referenz zu einem Symbol erfolgt, kann dieses nicht evaluiert werden. Die Evaluierung des Symbols erst dann, wenn dieses referenziert wurde, kann eine Latenzquelle sein. Ursache hierfür ist die Tatsache, dass Speicherseiten auf der Festplatte liegen können und Caches invalidiert werden können. Die Evaluierung von Symbolen vorab ist eine sicherere Vorgehensweise und hilft dabei, Latenzen zu verringern.

Die Auflösung von Symbolen beim Programmstart kann die Programminitialisierung etwas verlangsamen. Es vermeidet allerdings auch nicht-deterministische Latenzen während der Programmausführung, die durch den Aufruf von Symbolen verursacht werden können. Symbolauflösung beim Applikationsstart kann mittels der **LD_BIND_NOW**-Umgebungsvariable erfolgen. Die Einstellung von **LD_BIND_NOW** auf einen Wert ungleich Null fordert den Systemlader zum Aufruf aller nicht aufgelösten Symbole zum Zeitpunkt des Programmladens auf.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf der folgenden Handbuchseite:

» `ld.so(8)`

Teil III. Bibliotheksdienste

Systembefehle werden zur Änderung von Prioritäten, Prozessoraffinität und Scheduling-Richtlinien verwendet. Es ist auch möglich, diese Elemente aus Benutzerapplikationen heraus mithilfe von Bibliotheksfunktionen zu ändern.

Dieser Abschnitt erklärt, wie Prioritäten, Prozessoraffinität und Scheduler-Richtlinien mithilfe von Bibliotheksfunktionen geändert werden und wie die Ergebnisse dieser Änderungen geprüft werden können.

Kapitel 11. Einstellen des Schedulers

Es gibt zwei Möglichkeiten zur Einstellung und Beobachtung von Prozesskonfigurationen: Die Befehlszeilen-Hilfsprogramme und das grafische Tuna-Tool. Dieser Abschnitt verwendet die Befehlszeilen-Hilfsprogramme, aber alle Vorgänge können auch mit Tuna durchgeführt werden. Weitere Informationen zur Verwendung von Tuna finden Sie im *MRG Realtime Optimierungshandbuch* im Kapitel *Verwendung des Tuna-Interface*.

11.1. Verwendung von `chrt` zum Einstellen des Schedulers

Der Befehl **chrt** wird zur Überprüfung und Anpassung von Scheduler-Richtlinien und Prioritäten verwendet. Er kann neue Prozesse mit gewünschten Eigenschaften starten oder die Eigenschaften eines laufenden Prozesses ändern.

Um die Parameter eines bestimmten Prozesses zu überprüfen, verwenden Sie die **--pid** oder **-p**-Option allein, um die Prozess-ID (PID) anzugeben:

```
# chrt -p 468
pid 468's current scheduling policy: SCHED_FIFO
pid 468's current scheduling priority: 85

# chrt -p 476
pid 476's current scheduling policy: SCHED_OTHER
pid 476's current scheduling priority: 0
```

Um die Scheduling-Richtlinie eines Prozesses einzustellen, verwenden Sie die entsprechende Befehlsoption:

Tabelle 11.1. Richtlinienoptionen für den `chrt`-Befehl

Kurze Option	Lange Option	Beschreibung
-f	--fifo	Stellt Scheduler auf SCHED_FIFO ein
-o	--other	Stellt Scheduler auf SCHED_OTHER ein
-r	--rr	Stellt Scheduler auf SCHED_RR ein

Um die Priorität eines Prozesses einzustellen, geben Sie den Wert vor der PID des zu ändernden Prozesses an. Der folgende Befehl setzt den Prozess mit PID 1000 auf **SCHED_FIFO** mit einer Priorität von 50:

```
# chrt -f -p 50 1000
```

Der folgende Befehl setzt denselben Prozess (PID 1000) auf **SCHED_OTHER** mit einer Priorität von 0:

```
# chrt -o -p 0 1000
```

Um eine neue Applikation mit einer bestimmten Richtlinie und Priorität zu starten, geben Sie den Namen der Applikation (und den Pfad, falls nötig) zusammen mit den Parametern an. Der folgende Befehl startet **/bin/my-app** mit einer Richtlinie von **SCHED_FIFO** und Priorität 36:

```
# chrt -f 36 /bin/my-app
```



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf der folgenden Handbuchseite und in folgendem Buch:

- » *Tuna Benutzerhandbuch*
- » `chrt(1)`

11.2. Preemption

Ein Prozess kann *freiwillig* die CPU freigeben, da er entweder fertig ist oder auf ein Ereignis wartet (zum Beispiel auf Daten von der Festplatte, Tastatureingaben oder ein Netzwerkpaket).

Ein Prozess kann auch *unfreiwillig* zur Freigabe der CPU gezwungen werden. Dies wird als *Preemption* (oder auch *Prozessunterbrechung*) bezeichnet und erfolgt, wenn ein Prozess mit höherer Priorität die CPU verwenden will. Preemption kann besonders negative Auswirkungen auf die Leistung haben, und ständige Preemption kann zu einem als *Thrashing* bezeichneten Zustand führen, bei dem Prozesse laufend unterbrochen werden und keinem Prozess die vollständige Ausführung gestattet wird.

Um einen einzelnen Prozess auf freiwillige und unfreiwillige Preemption zu prüfen, sehen Sie sich die Inhalte von **/proc/PID/status** an, wobei **PID** die PID des Prozesses ist. Der folgende Befehl prüft die Preemption des Prozesses mit PID 1000:

```
# grep voluntary /proc/1000/status
voluntary_ctxt_switches: 194529
nonvoluntary_ctxt_switches: 195338
```

Das Ändern der Priorität einer Aufgabe kann dabei helfen, unfreiwillige Preemption zu verringern.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf der folgenden Handbuchseite und in folgendem Buch:

- » *Tuna Benutzerhandbuch*
- » `grep(1)`

11.3. Verwendung von Bibliotheksaufrufen zur Einstellung der Priorität

Bibliotheksaufrufe werden zur Einstellung der Priorität von nicht-Echtzeit-Prozessen verwendet. Diese sind:

- » **nice**
- » **getpriority**

► **setpriority**

Diese Funktionen operieren durch Abruf und Anpassung des *Nice-Werts* (etwa "Nettigkeitswert") eines Prozesses. Je *netter* ein Prozess ist, desto niedriger ist dessen Priorität. Die Senkung des Nice-Werts erhöht demnach die Priorität.

Echtzeitprozesse verwenden einen anderen Satz von Bibliotheksaufrufen zur Steuerung von Richtlinie und Priorität, worauf wir in diesem Abschnitt noch näher eingehen.



Wichtig

Die folgenden Funktionen erfordern alle den Einschluss einer **sched.h**-Header-Datei. Prüfen Sie stets die Rückgabecodes von Funktionen. Die entsprechenden Handbuchseiten liefern Informationen zu den verschiedenen verwendeten Codes.



Weitere Informationsquellen

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- nice(2)
- getpriority(2)
- setpriority(2)

11.3.1. sched_getscheduler

Die **sched_getscheduler()**-Funktion ruft die Scheduler-Richtlinie für eine bestimmte PID ab:

```
#include <sched.h>

int policy;

policy = sched_getscheduler(pid_t pid);
```

Die Symbole **SCHED_OTHER**, **SCHED_RR** und **SCHED_FIFO** sind ebenfalls in **sched.h** definiert. Sie können zur Prüfung der definierten Richtlinie oder zur Einstellung der Richtlinie verwendet werden:

```

#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main(int argc, char *argv[])
{
    pid_t pid;
    int policy;

    if (argc < 2)
        pid = 0;
    else
        pid = atoi(argv[1]);

    printf("Scheduler Policy for PID: %d -> ", pid);

    policy = sched_getscheduler(pid);

    switch(policy) {
        case SCHED_OTHER: printf("SCHED_OTHER
"); break;
        case SCHED_RR:    printf("SCHED_RR
"); break;
        case SCHED_FIFO:  printf("SCHED_FIFO
"); break;
        default:          printf("Unknown...
");
    }
}

```

11.3.2. sched_setscheduler

Die Scheduler-Richtlinie und andere Parameter können mithilfe der **sched_setscheduler()**-Funktion eingestellt werden. Derzeit besitzen Echtzeitrichtlinien einen Parameter, **sched_priority**. Dieser Parameter wird zur Anpassung der Prozesspriorität verwendet.

Die **sched_setscheduler**-Funktion erfordert drei Parameter in der Form:

sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);



Anmerkung

Die **sched_setscheduler(2)** Handbuchseite führt alle möglichen Rückgabewerte von **sched_setscheduler** auf, einschließlich der Fehlercodes.

Ist **pid** null, so wird die **sched_setscheduler()**-Funktion auf den aufrufenden Prozess angewendet.

Der folgende Codeausschnitt setzt die Scheduler-Richtlinie des aktuellen Prozesses auf **SCHED_FIFO** und die Priorität auf 50:

```

struct sched_param sp = { .sched_priority = 50 };
int ret;

ret = sched_setscheduler(0, SCHED_FIFO, &sp);
if (ret == -1) {
    perror("sched_setscheduler");
    return 1;
}

```

11.3.3. sched_getparam und sched_setparam

Die **sched_setparam()**-Funktion wird zur Einstellung der Scheduling-Parameter eines bestimmten Prozesses verwendet. Dies kann mittels **sched_getparam()**-Funktion verifiziert werden.

Anders als die **sched_getscheduler()**-Funktion, die nur die Scheduling-Richtlinie wiedergibt, gibt die **sched_getparam()**-Funktion alle Scheduling-Parameter für den angegebenen Prozess aus.

Der folgende Codeausschnitt liest die Priorität eines bestimmten Echtzeitprozesses und erhöht diese um zwei:

```

struct sched_param sp;
int ret;

/* reads priority and increments it by 2 */
ret = sched_getparam(0, &sp);
sp.sched_priority += 2;

/* sets the new priority */
ret = sched_setparam(0, &sp);

```



Anmerkung

Würde der Code oben in einer echten Applikation verwendet, so müsste er auch die Rückgabewerte der Funktion prüfen und eventuelle Fehler entsprechend handhaben.



Wichtig

Seien Sie vorsichtig mit der Erhöhung von Prioritäten. Das kontinuierliche Erhöhen um zwei in diesem Beispiel wird irgendwann zu einer ungültigen Priorität führen.

11.3.4. sched_get_priority_min und sched_get_priority_max

Die **sched_get_priority_min**- und **sched_get_priority_max**-Funktionen werden zur Prüfung des gültigen Prioritätsbereich für eine bestimmte Scheduler-Richtlinie verwendet.

Der einzig mögliche Fehler bei diesem Aufruf würde auftreten, falls die angegebene Scheduler-Richtlinie dem System unbekannt ist. In diesem Fall gibt die Funktion **-1** aus und **errno** ist auf **EINVAL** gesetzt.

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main()
{

    printf("Valid priority range for SCHED_OTHER: %d - %d\n",
           sched_get_priority_min(SCHED_OTHER),
           sched_get_priority_max(SCHED_OTHER));

    printf("Valid priority range for SCHED_FIFO: %d - %d\n",
           sched_get_priority_min(SCHED_FIFO),
           sched_get_priority_max(SCHED_FIFO));

    printf("Valid priority range for SCHED_RR: %d - %d\n",
           sched_get_priority_min(SCHED_RR),
           sched_get_priority_max(SCHED_RR));
}
```



Anmerkung

Bei **SCHED_FIFO** und **SCHED_RR** kann es sich um eine beliebige Zahl zwischen 1 und 99 handeln. POSIX berücksichtigt diesen Bereich nicht garantiert, portable Programme jedoch sollten diese Aufrufe verwenden.

11.3.5. sched_rr_get_interval

Die **SCHED_RR**-Richtlinie unterscheidet sich leicht von der **SCHED_FIFO**-Richtlinie. **SCHED_RR** weist nebenläufige Prozesse mit derselben Priorität mittels Round-Robin-Verfahren zu. Auf diese Weise wird jedem Prozess eine Zeitscheibe zugewiesen. Die **sched_rr_get_interval()**-Funktion meldet die Zeitscheibe, die jedem Prozess zugewiesen wurde.

Obwohl POSIX nur erfordert, dass diese Funktion mit **SCHED_RR**-Prozessen laufen muss, kann die **sched_rr_get_interval()**-Funktion die Länge der Zeitscheibe für jeden beliebigen Prozess unter Linux abrufen.

Die Informationen zur Zeitscheibe werden als **timespec** oder Anzahl an Sekunden oder Nanosekunden seit 00:00:00 GMT, 1 Januar 1970 wiedergegeben:

```
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
}
```

Die **sched_rr_get_interval**-Funktion benötigt die PID des Prozesses und ein "struct timespec":

```
#include <stdio.h>
#include <sched.h>

main()
{
    struct timespec ts;
    int ret;

    /* real apps must check return values */
    ret = sched_rr_get_interval(0, &ts);

    printf("Timeslice: %lu.%lu\n", ts.tv_sec, ts.tv_nsec);
}
```

Die folgenden Befehle führen das Testprogramm **sched_03** mit unterschiedlichen Richtlinien und Prioritäten aus. Prozesse mit einer **SCHED_FIFO**-Richtlinie geben eine Zeitscheibe von 0 Sekunden und 0 Nanosekunden wieder, was soviel wie "unbegrenzt" bedeutet:

```
$ chrt -o 0 ./sched_03
Timeslice: 0.38994072
```

```
$ chrt -r 10 ./sched_03
Timeslice: 0.99984800
```

```
$ chrt -f 10 ./sched_03
Timeslice: 0.0
```

Kapitel 12. Erstellen von Threads und Prozessen

Prozess- und Thread-Erstellung unterliegen der Systemauslastung und sind wesentlich für die Zuweisung von Ressourcen und CPU-Zeit. In einigen Szenarien ist eine Verzögerung zwischen Ereignis und dessen Bearbeitung akzeptabel. In den meisten Situationen jedoch führt dies zu unerwünschten und unnötigen Latenzen. Um dies zu verhindern, sollte der Pool von Prozessen oder Threads stets vorab erstellt werden, ehe diese von einem Dienst zur Bearbeitung einer Anfrage abgerufen werden. Für weitere Informationen siehe [Kapitel 4, *Threads und Prozesse*](#).

Kapitel 13. Mmap

Der **mmap**-Systemaufruf gestattet die Abbildung einer Datei (oder von Teilen einer Datei) im Speicher. Dies gestattet die Änderung der Dateiinhalte mithilfe einer Speicheroperation und vermeidet Systemaufrufe sowie Ein-/Ausgabeoperationen.

Synchronisieren Sie Änderungen stets wieder mit der Festplatte und berücksichtigen Sie einen möglicherweise hängenden Prozess, der zu Datenverlust führen könnten.



Anmerkung

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten und in folgendem Buch:

- » `mmap(2)`
- » *Linux System Programming* von Robert Love

Kapitel 14. Systemaufrufe

14.1. sched_yield

Die **sched_yield**-Funktion war ursprünglich darauf ausgelegt, einen Prozessor dazu zu veranlassen, einen anderen Prozess zu wählen als den derzeit laufenden. Diese Art von Aufruf ist jedoch fehleranfällig, wenn der Aufruf aus einer schlecht programmierten Applikation heraus ausgeführt wird.

Wird die **sched_yield()**-Funktion innerhalb von Prozessen mit Echtzeitprioritäten verwendet, so kann es zu unerwartetem Verhalten kommen. Der Prozess, der **sched_yield** aufgerufen hat, wird ans Ende der Warteschlange von Prozessen mit derselben Priorität verschoben. Geschieht dies in einer Situation, in der keine anderen Prozesse mit derselben Priorität laufen, so läuft der Prozess, der **sched_yield** aufgerufen hat, weiter. Ist die Priorität dieses Prozesses hoch, so kann es möglicherweise zu einem "Busy Loop" (einer aktiven Warteschleife) kommen, wodurch die Maschine nicht benutzbar wird.

Im Allgemeinen sollte **sched_yield** nicht auf Echtzeitprozessen verwendet werden.

14.2. getrusage()

Die **getrusage**-Funktion wird zum Abruf wichtiger Informationen von einem bestimmten Prozess oder dessen Threads verwendet. Dies liefert nicht alle verfügbaren Informationen, meldet jedoch Informationen zu Kontextwechseln und Seitenfehlern.

Kapitel 15. Timestamping

15.1. Hardware-Uhren

Multiprozessor-Systeme wie NUMA oder SMP haben mehrere Instanzen von Taktgebern. Die Art und Weise, wie diese Uhren interagieren und wie sie auf Systemereignisse wie z.B. CPU-Frequenzskalierung oder Eintritt in Energiesparmodi reagieren, entscheidet, ob sie als Taktgeber für den Realtime-Kernel geeignet sind.

Beim Systemstart erkennt der Kernel die verfügbaren Taktgeber und wählt einen davon zur Verwendung aus. Der bevorzugte Taktgeber ist der Time Stamp Counter (TSC); falls dieser nicht verfügbar ist, so ist der High Precision Event Timer (HPET) die zweitbeste Option. Allerdings verfügen nicht alle Systeme über HPET-Taktgeber, und einige HPET-Taktgeber können unzuverlässig sein.

Falls weder TSC noch HPET verfügbar sind, so sind andere mögliche Optionen der ACPI Power Management Timer (ACPI_PM), der Programmable Interval Timer (PIT) und die Real Time Clock (RTC). Die letzten beiden Optionen sind sehr aufwändig abzulesen bzw. haben eine sehr niedrige Auflösung (Zeitgranularität), sie sind deshalb für den Einsatz mit dem Realtime-Kernel nur bedingt geeignet.

In der `/sys/devices/system/clocksource/clocksource0/available_clocksource`-Datei sehen Sie eine Liste aller verfügbaren Taktgeber auf Ihrem System:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

In dem Beispiel oben sind die TSC, HPET und ACPI_PM Taktgeber verfügbar.

Den derzeit verwendeten Taktgeber finden Sie in der `/sys/devices/system/clocksource/clocksource0/current_clocksource`-Datei:

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Es ist möglich, einen anderen Taktgeber aus der Liste in der `/sys/devices/system/clocksource/clocksource0/available_clocksource`-Datei auszuwählen. Schreiben Sie dazu den Namen der Uhr in die `/sys/devices/system/clocksource/clocksource0/current_clocksource`-Datei. Der folgende Befehl legt beispielsweise HPET als den zu verwendenden Taktgeber fest:

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```



Wichtig

Der Kernel wählt aus den verfügbaren Taktgebern den besten aus. Diese Auswahl zu überschreiben, wird in der Regel nicht empfohlen, es sei denn, Sie sind sich über die Auswirkungen im Klaren.

Obwohl TSC generell der bevorzugte Taktgeber ist, hat er doch mehrere Nachteile. Beispielsweise kann die TSC-Uhr stehen bleiben, wenn das System in einen Ruhezustand eintritt, oder sie kann allmählich nachgehen, wenn ihre CPUs in tiefere C-Zustände (Energiesparzustände) eintritt oder Operationen zur Geschwindigkeits- oder Frequenzskalierung durchführt.

Sie können diese Nachteile vom TSC jedoch umgehen, indem Sie zusätzliche Kernel-Boot-Parameter konfigurieren. Beispielsweise hindert der ***idle=poll*** Parameter die Uhr am Eintritt in den Ruhezustand, und der ***processor.max_cstate=1*** Parameter hindert die Uhr am Eintritt in tiefe C-Zustände. Beachten Sie jedoch, dass in beiden Fällen der Energieverbrauch steigt, da das System dauerhaft auf höchster Geschwindigkeit läuft.



Anmerkung

Eine umfassende Liste der Taktgeber finden Sie im Kapitel *Timing Measurements* in dem Buch *Understanding The Linux Kernel* von Daniel P. Bovet und Marco Cesati.

15.1.1. Ablesen von Hardware-Uhren

Den TSC abzulesen bedeutet im Wesentlichen, ein Register vom Prozessor zu lesen. Die HPET-Uhr abzulesen bedeutet, einen Speicherbereich zu lesen. Das Ablesen des TSC ist schneller und bietet somit einen deutlichen Leistungsvorteil, wenn Tausende von Nachrichten pro Sekunde mit einem Timestamp versehen werden müssen.

Mithilfe eines einfachen Programms, das den aktuellen Taktgeber 10.000.000 Mal nacheinander abliest, können Sie beobachten, wie lange es dauert, die verfügbaren Taktgeber abzulesen:

Beispiel 15.1. Vergleich des Zeitaufwands zum Ablesen der Hardware-Uhren

In diesem Beispiel ist der derzeit verwendete Taktgeber TSC, wie die Ausgabe des **cat**-Befehls zeigt. Mithilfe des **time**-Befehls können Sie beobachten, wie lange es dauert, den Taktgeber 10 Millionen Mal abzulesen.

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
# time ./clock_timing

real 0m0.601s
user 0m0.592s
sys 0m0.002s
```

Anschließend wird der Taktgeber auf HPET geändert, um zu vergleichen, wie lange es mit diesem Taktgeber dauert, 10 Millionen Timestamps zu generieren:

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
hpet
# time ./clock_timing

real 0m12.263s
user 0m12.197s
sys 0m0.001s
```

Dieselben Schritte werden schließlich mit dem ACPI_PM Taktgeber wiederholt:

```
# echo acpi_pm >
/sys/devices/system/clocksource/clocksource0/current_clocksource
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
acpi_pm
# time ./clock_timing

real 0m24.461s
user 0m0.504s
sys 0m23.776s
```

Die **time(1)** Handbuchseite liefert detaillierte Informationen über die Verwendung des Befehls und die Interpretation der Ergebnisse. Das Beispiel oben betrachtet die folgenden Kategorien:

- ▶ **real**: Die Gesamtzeit ab Programmaufruf bis zum Abschluss des Prozesses. **real** umfasst die **user**- und **sys**-Zeiten und ist in der Regel größer als die Summe der beiden. Falls dieser Prozess von einer Applikation mit höherer Priorität oder durch ein Systemereignis wie einen Hardware-Interrupt (IRQ) unterbrochen wird, so fließt diese Wartezeit ebenfalls in den **real**-Wert ein.
- ▶ **user**: Die Zeit, die der Prozess im User Space verbringt, um Aufgaben auszuführen, die keinen Eingriff vom Kernel erforderten.
- ▶ **sys**: Die Zeit, die der Kernel damit verbringt, die vom Benutzerprozess erforderlichen Aufgaben auszuführen. Zu diesen Aufgaben gehört das Öffnen von Dateien, das Lesen und Schreiben von Dateien oder I/O-Ports, Speicherzuweisung, Thread-Erstellung und Netzwerkoperationen.

Wie die Ergebnisse von [Beispiel 15.1, „Vergleich des Zeitaufwands zum Ablesen der Hardware-Uhren“](#) belegen, ist die Effizienz der Timestamp-Generierung in absteigender Reihenfolge: TSC, HPET, ACPI_PM. Die Ursache liegt in dem zunehmenden Overhead, der beim Abruf der Zeitwerte von den

HPET und ACPI_PM Uhren entsteht.

15.2. POSIX-Uhren

POSIX umfasst einen Standard zur Implementierung und Darstellung von Zeitquellen. Im Gegensatz zur Hardware-Uhr, die vom Kernel ausgewählt und systemweit implementiert wird, kann die POSIX-Uhr von jeder Applikation separat ausgewählt werden, ohne Auswirkungen auf andere Applikationen im System.

- » **CLOCK_REALTIME**: repräsentiert die Uhrzeit in der realen Welt, auch oft "tatsächliche Zeit" genannt. Diese Uhr wird beim Generieren von Timestamps für Ereignisse und bei der Interaktion mit dem Benutzer verwendet. Sie kann von einem Benutzer mit den entsprechenden Berechtigungen verändert werden. Allerdings ist bei Änderungen durch den Benutzer Vorsicht geboten, da es zu fehlerhaften Daten führen kann, wenn der Wert der Uhr zwischen zwei Ableseoperationen verändert wurde.
- » **CLOCK_MONOTONIC**: repräsentiert die monoton fortschreitende Zeit seit dem Systemstart. Diese Uhr kann von keinem Prozess verändert werden und ist der bevorzugte Zeitgeber zum Messen von Zeitabständen zwischen Ereignissen. Die nachfolgenden Beispiele in diesem Abschnitt nutzen **CLOCK_MONOTONIC** als POSIX-Uhr.



Anmerkung

Mehr Informationen über POSIX-Uhren finden Sie auf der folgenden Handbuchseite und im folgenden Buch:

- » `clock_gettime()`
- » *Linux System Programming* von Robert Love

Die Funktion zum Ablesen einer POSIX-Uhr ist **clock_gettime()**, definiert unter **<time.h>**. Der **clock_gettime()**-Befehl akzeptiert zwei Parameter: die ID der POSIX-Uhr und eine timespec-Struktur, die mit der Zeitspanne gefüllt wird, die zum Ablesen der Uhr nötig ist. Das folgende Beispiel zeigt die Funktion, mit der der Zeitaufwand zum Ablesen der Uhr gemessen wird:

Beispiel 15.2. Verwenden von `clock_gettime()` zum Messen des Zeitaufwands zum Ablesen von POSIX-Uhren

```
#include <time.h>

main()
{
    int rc;
    long i;
    struct timespec ts;

    for(i=0; i<100000000; i++) {
        rc = clock_gettime(CLOCK_MONOTONIC, &ts);
    }
}
```

Sie können das obige Beispiel weiter verbessern, indem Sie beispielsweise Funktionen einfügen, um den Wiedergabewert von **clock_gettime()** zu prüfen, um den Wert der **rc**-Variable zu prüfen, oder

um zu gewährleisten, dass dem Inhalt der **ts**-Struktur vertraut werden kann. Auf der **clock_gettime()**-Handbuchseite finden Sie weitere Informationen über das Schreiben zuverlässigerer Applikationen.



Wichtig

Programme, die die **clock_gettime()**-Funktion nutzen, müssen mit der **rt**-Bibliothek verknüpft werden, indem **'-lrt'** zur **gcc**-Befehlszeile hinzugefügt wird.

```
cc clock_timing.c -o clock_timing -lrt
```

15.2.1. CLOCK_MONOTONIC_COARSE und CLOCK_REALTIME_COARSE

Die Funktionen wie z.B. **clock_gettime()** und **gettimeofday()** haben eine Entsprechung im Kernel in Form eines Systemaufrufs. Wenn der Benutzerprozess **clock_gettime()** aufruft, dann ruft die entsprechende C-Bibliothek (**glibc**) den **sys_clock_gettime()**-Systemaufruf auf, der die angeforderte Operation ausführt und das Ergebnis an das Benutzerprogramm zurückgibt.

Dieser Kontextwechsel von der Benutzerapplikation zum Kernel ist jedoch mit Aufwand verbunden. Zwar ist dieser Aufwand an sich gering; falls die Operation jedoch tausendfach wiederholt wird, kann sich der kumulierte Aufwand auf die Leistung der Applikation insgesamt niederschlagen.

Um den Kontextwechsel zum Kernel zu vermeiden und somit das Ablesen der Uhr zu beschleunigen, wurde Unterstützung für die **CLOCK_MONOTONIC_COARSE** und **CLOCK_REALTIME_COARSE** POSIX-Uhren in Form einer VDSO-Bibliotheksfunktion hinzugefügt. Die **_COARSE**-Varianten sind schneller abzulesen und haben eine Genauigkeit (Auflösung) von einer Millisekunde (ms).

15.2.2. Verwenden von clock_getres() zum Vergleich der Zeitauflösung

Mithilfe der **clock_getres()**-Funktion können Sie die Auflösung einer POSIX-Uhr prüfen.

clock_getres() nutzt dieselben zwei Parameter wie **clock_gettime()**: die ID der POSIX-Uhr und einen Verweis auf die **timespec**-Struktur, in der das Ergebnis ausgegeben wird. Die folgende Funktion ermöglicht es Ihnen, die Auflösung von **CLOCK_MONOTONIC** und **CLOCK_MONOTONIC_COARSE** miteinander zu vergleichen:

```
main()
{
    int rc;
    struct timespec res;

    rc = clock_getres(CLOCK_MONOTONIC, &res);
    if (!rc)
        printf("CLOCK_MONOTONIC: %ldns\n", res.tv_nsec);
    rc = clock_getres(CLOCK_MONOTONIC_COARSE, &res);
    if (!rc)
        printf("CLOCK_MONOTONIC_COARSE: %ldns\n", res.tv_nsec);
}
```

Beispiel 15.3. Beispielausgabe von `clock_getres`

```
TSC:
# ./clock_resolution
CLOCK_MONOTONIC: 1ns
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)

HPET:
# ./clock_resolution
CLOCK_MONOTONIC: 1ns
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)

ACPI_PM:
# ./clock_resolution
CLOCK_MONOTONIC: 1ns
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)
```

15.2.3. Verwenden von C-Code zum Vergleich der Zeitauflösung

Mithilfe des folgenden Codes ist es möglich, das Format der Daten zu beobachten, die von der **CLOCK_MONOTONIC** POSIX-Uhr gelesen werden. Alle neun Ziffern im **tv_nsec**-Feld der `timespec`-Struktur sind bedeutsam, da die Uhr eine Nanosekundenauflösung hat. Die Beispielfunktion namens **clock_test.c** sieht folgendermaßen aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
    int i;
    struct timespec ts;

    for(i=0; i<5; i++) {
        clock_gettime(CLOCK_MONOTONIC, &ts);
        printf("%ld.%ld\n", ts.tv_sec, ts.tv_nsec);
        usleep(200);
    }
}
```

Beispiel 15.4. Beispielausgabe von `clock_test.c` und `clock_test_coarse.c`

Wie im Code oben festgelegt, liest die Funktion die Uhr fünf Mal im Abstand von jeweils 200 Mikrosekunden:

```
# cc clock_test.c -o clock_test -lrt
# ./clock_test
218449.986980853
218449.987330908
218449.987590716
218449.987849549
218449.988108248
```

Nehmen Sie nun denselben Quellcode, benennen ihn um in `clock_test_coarse.c` und ersetzen **CLOCK_MONOTONIC** durch **CLOCK_MONOTONIC_COARSE**. Das Ergebnis sieht dann in etwa folgendermaßen aus:

```
# ./clock_test_coarse
218550.844862154
218550.844862154
218550.844862154
218550.845862154
218550.845862154
```

Die **_COARSE** Uhren haben eine Auflösung von einer Millisekunde, daher sind nur die ersten drei Ziffern des `tv_nsec`-Felds in der `timespec`-Struktur von Bedeutung. Das obige Ergebnis kann also folgendermaßen gelesen werden:

```
# ./clock_test_coarse
218550.844
218550.844
218550.844
218550.845
218550.845
```

Die **_COARSE** Varianten der POSIX-Uhren sind insbesondere in Situationen hilfreich, in denen Timestamping mit Millisekundenauflösung durchgeführt werden kann. Die Vorteile sind auf jenen Systemen deutlicher, die bislang Hardware-Uhren mit hohem Aufwand für Ableseoperationen verwendeten, wie z.B. ACPI_PM.

15.2.4. Verwenden des `time`-Befehls zum Vergleich des Aufwands für Ableseoperationen

Sie können den Aufwand für Ableseoperationen der **CLOCK_MONOTONIC** und **CLOCK_MONOTONIC_COARSE** Varianten der verfügbaren Hardware-Uhren miteinander vergleichen, indem Sie den `time`-Befehl verwenden, um den Taktgeber 10 Millionen Mal nacheinander abzulesen. Das folgende Beispiel verwendet die TSC, HPET und ACPI_PM Hardware-Uhren. Weitere Informationen über die Interpretation der Ausgabe des `time`-Befehls finden Sie in [Abschnitt 15.1.1, „Ablesen von Hardware-Uhren“](#).

Beispiel 15.5. Vergleich des Zeitaufwands zum Ablesen von POSIX-Uhren

```
TSC:
# time ./clock_timing_monotonic

real 0m0.567s
user 0m0.559s
sys 0m0.002s

# time ./clock_timing_monotonic_coarse

real 0m0.120s
user 0m0.118s
sys 0m0.001s

HPET:
# time ./clock_timing_monotonic

real 0m12.257s
user 0m12.179s
sys 0m0.002s

# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.118s
sys 0m0.000s

ACPI_PM:
# time ./clock_timing_monotonic

real 0m25.524s
user 0m0.451s
sys 0m24.932s

# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.117s
sys 0m0.001s
```

Wie [Beispiel 15.5, „Vergleich des Zeitaufwands zum Ablesen von POSIX-Uhren“](#) zeigt, ist die **sys**-Zeit (die Zeit, die der Kernel damit verbringt, die vom Benutzerprozess erforderlichen Aufgaben auszuführen) erheblich geringer, wenn **_COARSE**-Uhren verwendet werden. Dies wird besonders deutlich bei den Werten der ACPI_PM-Uhr, was darauf hindeutet, dass die **_COARSE**-Varianten von POSIX-Uhren im Vergleich zu Uhren mit hohem Ableseaufwand erhebliche Leistungsvorteile bieten.

Kapitel 16. Weitere Informationen

16.1. Melden von Fehlern

Fehlerdiagnose

Bevor Sie eine Fehlermeldung erstatten, befolgen Sie diese Schritte, um festzustellen, wo die eigentliche Fehlerursache liegt. Dies ist überaus hilfreich bei der Fehlerbehebung.

1. Stellen Sie sicher, dass Sie die aktuellste Version des Red Hat Enterprise Linux 6 Kernels verwenden, und booten Sie diesen vom "grub"-Menü. Versuchen Sie dann, das Problem mit dem Standard-Kernel zu reproduzieren. Falls das Problem nach wie vor auftritt, melden Sie den Fehler für Red Hat Enterprise Linux 6, *NICHT* für MRG Realtime.
2. Falls das Problem bei Verwendung des Standard-Kernels nicht auftritt, so ist der Fehler wahrscheinlich ein Ergebnis späterer Änderungen in entweder:
 - a. dem Upstream-Kernel, auf dem MRG Realtime basiert. Zum Beispiel basiert Red Hat Enterprise Linux 6 auf Kernel-Version 2.6.32 und MRG Realtime basiert auf 3.2
 - b. MRG Realtime spezifischen Erweiterungen, die Red Hat zusätzlich zum Grund-Kernel (3.2) angewendet hat

Um das Problem näher zu bestimmen, versuchen Sie, das Problem an einem unbearbeiteten Upstream 3.2 Kernel reproduzieren können. Aus diesem Grund liefern wir neben dem MRG Realtime Kernel auch eine **vanilla**-Kernelvariante. Beim **vanilla**-Kernel handelt es sich um einen Upstream Kernel-Build ohne die MRG Realtime Features.

Melden eines Fehlers

Wenn Sie sicher sind, dass der Fehler spezifisch für MRG Realtime ist, folgen Sie diesen Anweisungen, um den Fehler zu melden:

1. Erstellen Sie einen [Bugzilla](#)-Benutzerkonto.
2. Melden Sie sich an und klicken Sie auf [Enter A New Bug Report](#).
3. Sie müssen nun das Produkt angeben, in dem der Fehler auftritt. MRG Realtime erscheint in der Red Hat Produktliste unter **Red Hat Enterprise MRG**. Es ist wichtig, dass Sie das richtige Produkt angeben, in dem der Fehler auftritt.
4. Fahren Sie mit der Eingabe der Fehlerinformationen fort, indem Sie die entsprechende Komponente zuordnen und eine ausführliche Problembeschreibung liefern. Bei Eingabe der Problembeschreibung sollten Sie auch mitteilen, ob Sie das Problem mit dem standardmäßigen Red Hat Enterprise Linux 6 oder dem bereitgestellten **vanilla**-Kernel reproduzieren konnten.

16.2. Weitere Informationsquellen

- Red Hat Enterprise MRG Produktinformation
 - <http://www.redhat.com/mrg>
- MRG Realtime Optimierungshandbuch und andere Red Hat Enterprise MRG Dokumentation
 - http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/
- Mailing-Liste
 - Um auf dieser Liste zu posten, schicken Sie eine E-Mail an rhemrg-users-list@redhat.com
 - Die Mailing-Liste können Sie unter <https://www.redhat.com/mailman/listinfo/rhemrg-users-list> abonnieren.

Versionsgeschichte

Version 3-1.400	2013-10-31	Rüdiger Landmann
Rebuild with publican 4.0.0		
Version 3-0	Thu May 31 2012	Cheryn Tan
Vorbereitet für Veröffentlichung (MRG 2.2).		
Version 2-5	Tue May 15 2012	Cheryn Tan
BZ#813989 - Timestamping-Kapitel überarbeitet.		
Version 2-4	Wed May 9 2012	Cheryn Tan
BZ#813989 - Kapitel 15 über Timestamping hinzugefügt.		
Version 2-2	Tue Apr 10 2012	Cheryn Tan
Fehler aus Docs-QE behoben, Kernel-Hinweise aktualisiert, Hinweise auf RHEL 5 entfernt.		
Version 2-1	Tue Feb 28 2012	Tim Hildred
Konfigurationsdatei für neues Publikationstool.		
Version 2-0	Wed Dec 7 2011	Alison Young
Vorbereitet für Veröffentlichung		
Version 1-6	Wed Nov 16 2011	Alison Young
BZ#752406 - RHEL-Versionen geändert		
Version 1-2	Wed Oct 5 2011	Alison Young
BZ#712267 - Link zu nicht vorhandener Mailingliste		
Version 1-1	Thu Sep 22 2011	Alison Young
Versionsnummerierung geändert		
Version 1-0	Thu Jun 23 2011	Alison Young
Vorbereitet für Veröffentlichung		
Version 0.1-3	Thu May 19 2011	Alison Young
Änderungen nach technischer Überprüfung		
Version 0.1-2	Mon May 16 2011	Alison Young
BZ#666961 - Aktualisierung für RHEL6		
Version 0.1-1	Tue Apr 05 2011	Alison Young
BZ#683586 - Abschnitt über weitere Informationsquellen aktualisiert Kleinere XML-Aktualisierungen		
Version 0.1-0	Wed Feb 23 2011	Alison Young
Abgezweigt von 1.3		